Abstraction for Crash-Resilient Objects (Extended Version)

Artem Khyzha* and Ori Lahav

Tel Aviv University, Israel

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.

1 Introduction

Non-volatile memory (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., [12,29]). Recently, NVM has started to become available in commodity architectures of manufacturers such as Intel and ARM [4, 21], and formal (operational and declarative) models of these systems have been proposed [9, 23, 28].

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 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.

* — Now at Arm Ltd, UK

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 the guaranteed 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 for providing means to reason about client programs.

In this paper, we formulate and prove an abstraction theorem for concurrent programs utilizing non-volatile memory. We target the PSC ("Persistent Sequential Consistency") model of [23], 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, [23] presented compiler mappings from PSC to the x86 persistency model of [28], which can be used to ensure PSC semantics on Intel machines.

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 [19], 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, 11].

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 [17], and durable linearizability and its buffered variant [22]. 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., [16, 18, 24]), and was applied before for deriving abstraction theorems in different contexts [8, 14, 15]. 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 specification code has several advantages over correctness notions based on sequential specifications of libraries. First, specifications and implementations are expressed and reasoned about in a unified framework, alleviating the need to interpret the use of sequentially specified code 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 similarly needs a way to govern the per-

sistence offering intuitive specifications for libraries and simpler reasoning about their clients. 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 for the writes they protect.

For example, when recovering after a crash during a run of the tiny program $\dot{\mathtt{x}} := 1 \,;\, \dot{\mathtt{y}} := 1$, 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 $\dot{\mathtt{x}} \in \{0,1\} \wedge \dot{\mathtt{y}} \in \{0,1\}$.[1] In turn, if a persistence block is used, as in the program $\mathtt{beginPB}(\dot{\mathtt{x}}, \dot{\mathtt{y}}) \,;\, \dot{\mathtt{x}} := 1 \,;\, \dot{\mathtt{y}} := 1 \,;\, \mathtt{endPB}(\dot{\mathtt{x}}, \dot{\mathtt{y}})$, 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 [20] (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 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) compares the histories produced by using a library L to those produced by its specification $L^{\#}$ for 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 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 ensures that any client can communicate with the library only via values passed to and returned from method invocations, and these interactions are fully captured in the library's histories. (This restriction can be alleviated to allow ownership transfer following [15]).

[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.

However, under NVM semantics, 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 $\texttt{fo}(\cdot)$). 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 $\dot{\texttt{x}} := 1\,;\,\texttt{fo}(\dot{\texttt{x}})\,;\,\texttt{sfence}$, 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 $\dot{\texttt{x}} := 1\,;\,\texttt{fo}(\dot{\texttt{x}})\,;\,\texttt{call}(f)\,;\,\dot{\texttt{y}} := 1$ that uses $L^{\#}$, we have $\dot{\texttt{y}} = 1 \implies \dot{\texttt{x}} = 1$ as a global invariant: if the system has crashed and we have $\dot{\texttt{y}} = 1$ in the NVM, then the `sfence` ensures that $\dot{\texttt{x}} = 1$ is in the NVM as well. Nevertheless, due to out-of-order persistence, if we use L in this program we may get $\dot{\texttt{y}} = 1 \wedge \dot{\texttt{x}} = 0$ after a crash. Now, the histories that the library L may produce for the MGC are all (well-formed) sequences of "call" and "return" transitions, which are exactly the same histories that $L^{\#}$ produces. Thus, when inspecting histories of L and of $L^{\#}$, we do not have sufficient information to see 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 another instruction 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.We believe that most libraries do not intend to provide a store fence functionality to their clients, and 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 (called "CLFLUSH" in [21]).

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 store fence in the library's specification (rather than the localized counterpart), and

the abstraction theorem has to handle store fences with their standard global semantics. To do so, we expose (global) store fences in histories (together with method invocations and responses). 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 store fence whenever $L^{\#}$ does (which does not hold for the example above). Our history notion 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 of L. An interesting compositionality question arises: Are we allowed to assume the library's specification when checking if 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 policy adherence when using $L^{\#}$. 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

[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.

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; see [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.

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 sequences (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 following table summarizes the domains that we use in the technical development and indicates the metavariables we use to range over each domain:

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

Thus, there are three kinds of variables: shared non-volatile, shared volatile, and thread-local ones (called registers), which are also volatile. The distinguished

method 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:

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

Expressions are constructed with arithmetic and boolean operations over registers and values. Instructions consist of a local assignment $r := e$; a conditional $\texttt{if } e \texttt{ goto } n_1 \shortmid \ldots \shortmid n_m$ for non-deterministically jumping to a program counter from $\{n_1, \ldots, n_m\}$ when e evaluates to non-zero or, otherwise, skipping; ($\texttt{goto } n_1 \shortmid \ldots \shortmid n_m$ encoded as $\texttt{if } 1 \texttt{ goto } n_1 \shortmid \ldots \shortmid n_m$); havoc for arbitrarily modifying all registers; a write to memory $x := e$; a read from memory $r := x$; and two atomic read-modify-write instructions (RMWs), a fetch-and-add $r := \texttt{FADD}(x, e)$ and a compare-and-swap $r := \texttt{CAS}(x, e, e)$. The former loads the value from a variable x into r and increments the value in memory. The latter also loads the value from a variable x into r, but overwrites it with the second expression in case if the loaded value coincides with the first expression. There are also several types of explicit persist instructions: a *flush* instruction $\texttt{fl}(\dot{x})$ and its optimized version $\texttt{fo}(\dot{x})$, called *flush-optimal* ([21] refers to them as CLFLUSH and CLFLUSHOPT). The store fence instruction sfence enforces ordering on persistence of writes.

We extend this standard instruction set to support calling and specifying library methods. There is a call instruction $\texttt{call}(f)$ and a return instruction return. There is also a *local store fence* instruction $\texttt{lsfence}(\dot{X})$, which is a fictional instruction relaxing the semantics of sfence by only enforcing the persistence ordering for the given set of variables $\dot{X}$ (thus, $\texttt{lsfence}(\mathsf{NVVar})$ is equivalent to sfence, and $\texttt{fl}(\dot{x})$ is equivalent to $\texttt{fo}(\dot{x})\,;\texttt{lsfence}(\{\dot{x}\})$). Finally, there are instructions to begin and end a *persistence block*, $\texttt{beginPB}(\dot{X})$ and $\texttt{endPB}(\dot{X})$, respectively. The persistence block is a special construct we use to demark the writes that need to persist simultaneously after the block ends, either non-deterministically or triggered by a flush for 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 $f \in \{\mathsf{main}\} \uplus \mathsf{F}$. 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 $Pr(\mathtt{T}_1)(f)$.

3.2 Program Semantics

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

Definition 1 (Labeled transition systems). A *labeled transition system* (LTS, for short) A is a tuple $\langle \Sigma, Q, q_{\mathsf{Init}}, T \rangle$, 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 $A.\boldsymbol{\Sigma}$, $A.\mathtt{Q}$, $A.\mathsf{q}_{\mathsf{Init}}$ 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 $t \in A.\boldsymbol{\Sigma}^*$, we write $\xrightarrow{t}_A$ for the composition $\xrightarrow{t(1)}_A ; \ldots ; \xrightarrow{t(|t|)}_A$. A sequence $t \in A.\boldsymbol{\Sigma}^*$ such that $A.\mathsf{q}_{\mathsf{Init}} \xrightarrow{t}_A q$ 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 $A.\mathsf{q}_{\mathsf{Init}} \xrightarrow{t}_A q$ 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* $\mathtt{R}(x, v_{\mathtt{R}})$	a *flush* $\mathtt{FL}(\dot{x})$	a *read-modify-write* $\mathtt{RMW}(x, v_{\mathtt{R}}, v_{\mathtt{W}})$
a *write* $\mathtt{W}(x, v_{\mathtt{W}})$	a *flush-opt* $\mathtt{FO}(\dot{x})$	a *failed CAS* $\mathtt{R\text{-}ex}(x, v_{\mathtt{R}})$
a *call* $\mathtt{CALL}(f, \phi)$	an *sfence* $\mathtt{SF}$	a *persistent-block start* $\mathsf{beginPB}(\dot{X})$
a *return* $\mathtt{RET}(f, \phi)$	a *local sfence* $\mathtt{LSF}(\dot{X})$	a *persistent-block end* $\mathsf{endPB}(\dot{X})$

where $x \in \mathsf{Var}$, $v_{\mathtt{R}}, v_{\mathtt{W}} \in \mathsf{Val}$, $\dot{x} \in \mathsf{NVVar}$, $\dot{X} \subseteq \mathsf{NVVar}$, $f \in \mathsf{F}$, and $\phi : \mathsf{Reg} \to \mathsf{Val}$. We denote by Lab the set of all action labels. The functions $\mathtt{typ}$, $\mathtt{var}$, $\mathtt{val}_{\mathtt{R}}$, $\mathtt{val}_{\mathtt{W}}$ retrieve (when applicable) the type ($\mathtt{R}/\mathtt{W}/\mathtt{RMW}/\ldots$), variable ($x$ or $\dot{x}$), read value ($v_{\mathtt{R}}$), and written value ($v_{\mathtt{W}}$) of an action label. We write $\mathtt{varset}(l)$ for the (possibly empty) set of all variables mentioned in l (e.g., $\mathtt{varset}(\mathtt{R}(x, v_{\mathtt{R}})) = \{x\}$, $\mathtt{varset}(\mathtt{LSF}(\dot{X})) = \dot{X}$, and $\mathtt{varset}(\mathtt{SF}) = \emptyset$).

Action labels correspond to the different interactions that a program may have with the memory system. Most of them are in one-to-one correspondence with the instructions of the language. Fetch-and-add and successful compare-and-swap instructions are captured here by a general RMW label ($\mathtt{RMW}(x, v_{\mathtt{R}}, v_{\mathtt{W}})$), while failed compare-and-swaps (which did not read the expected value) correspond to special read label $\mathtt{R\text{-}ex}(x, v_{\mathtt{R}})$, which allows us to distinguish such transitions from plain reads and provide them with stronger guarantees.

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 $\phi_{\mathsf{Init}} \stackrel{\mathsf{def}}{=} \lambda r.\, 0$.
- The transitions are formally defined in the suplementary material.

The transitions straightforwardly describe changes in control flow and register store. A local assignment $r := e$ updates a register r with the value of an expression e. An havoc instruction `havoc` arbitrarily replaces the register store. A conditional `if` e `goto` $n_1 \mid ... \mid n_m$ tests whether e evaluates to non-zero, in which case it updates the program counter to any n_i, or increments it otherwise.

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. The fetch-and-add $r := \texttt{FADD}(x, e)$ loads an arbitrary value into the destination register r, announcing in the label $\texttt{RMW}(x, v, v + \phi(e))$ that the value loaded is v and the value stored is $v + \phi(e)$. The two transitions for the compare-and-swap $r := \texttt{CAS}(x, e_{\texttt{R}}, e_{\texttt{W}})$ both load an arbitrary value v into the destination register r, except the successful compare-and-swap transition announces in the label $\texttt{RMW}(x, \phi(e_{\texttt{R}}), \phi(e_{\texttt{W}}))$ that $\phi(e_{\texttt{R}})$ is the value loaded and $\phi(e_{\texttt{W}})$ is the value stored, while the failed compare-and-swap announces in the label $\texttt{R-ex}(x, v)$ that the value loaded that is different from $\phi(e_{\texttt{R}})$. Flush, flush-optimal, sfence, local sfence, and persistence-block-begin/end transitions all act as no-ops, and simply announce themselves in the transition label, using the function $\mathsf{matching\_label}$ that maps each instruction to its label ($\texttt{fl}(\dot{x}) \mapsto \texttt{FL}(\dot{x}), \texttt{fo}(\dot{x}) \mapsto \texttt{FO}(\dot{x})$ and so on). Thus, the execution of all these instructions is only constrained once program semantics is synchronized with the memory system semantics via announced transition labels.

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

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:
  - $\langle pc, \phi \rangle \in \mathbb{N} \times (\mathsf{Reg} \to \mathsf{Val})$ stores the state of the instruction 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 $q.\texttt{pc}$, $q.\phi$, $q.\texttt{pc}_{\mathsf{s}}$, and $q.\texttt{f}$ the components of a state $q \in S.\texttt{Q}$.

- The initial state is $\langle 0, \phi_{\mathsf{Init}}, \bot, \mathsf{main}\rangle$.
- The transitions are given by:

$$\textsc{normal}\ \frac{f \in \{\mathsf{main}\} \cup \mathsf{F} \qquad \langle pc, \phi\rangle \xrightarrow{l_\epsilon}_{S(f)} \langle pc', \phi'\rangle}{\langle pc, \phi, pc_{\mathtt{s}}, f\rangle \xrightarrow{l_\epsilon}_{S} \langle pc', \phi', pc_{\mathtt{s}}, f\rangle} \qquad \textsc{call}\ \frac{S(\mathsf{main})(pc) = \mathtt{call}(f) \qquad l = \mathtt{CALL}(f, \phi)}{\langle pc, \phi, \bot, \mathsf{main}\rangle \xrightarrow{l}_{S} \langle 0, \phi, pc+1, f\rangle}$$

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

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. In turn, it is particularly useful for simplifying the library correctness condition that only considers inclusion of histories (see §5). For instance, referring back to the example in §2.2, the library implementing f using `sfence` should be considered a refinement of the one that simply returns (here, we switched the roles of L and $L^{\#}$ from §2.2). The NON-DET-SFENCE transition allows us to see this in the libraries' histories. Indeed, the no-op specification may perform a non-deterministic sfence in its histories that match the ones executed by the `sfence` instruction in the concrete implementation.

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: $\overline{q}_{\mathsf{Init}} \stackrel{\text{def}}{=} \langle Pr(\mathtt{T}_1).\mathsf{q}_{\mathsf{Init}}, \ldots, Pr(\mathtt{T}_{\mathsf{N}}).\mathsf{q}_{\mathsf{Init}}\rangle$.
- The transitions are either interleaved thread transitions or crash transitions reinitializing the program state, and are given by:

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

4 The PSC Memory System

In this section, we present PSC ("Persistent Sequential Consistency"), the persistency model from [23], which we use as the memory system. We follow its op-

erational presentation as an LTS with non-deterministic memory-internal transitions that flush stores from the volatile part to the non-volatile part.

We first introduce PSC as it is in [23] (extended with standard volatile memory alongside with the non-volatile one). In §4.1, we present the extensions added in this paper that are useful for library abstraction. In §4.2, we define the synchronization of programs with the PSC memory system. Finally, in §4.3, we establish certain separation properties of PSC that are essential in our proof.

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 issuing the barrier. The effect of flush-optimal is then delayed until the same thread performs an sfence (store fence), 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.

Formally, 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

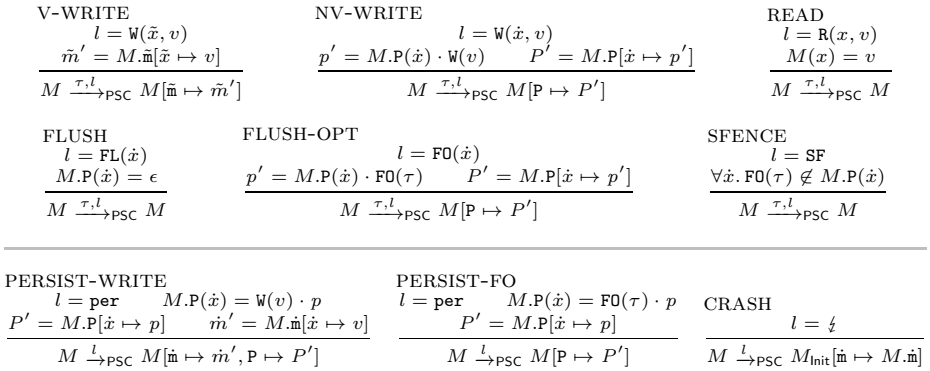

Fig. 1. Selected transitions of PSC (see supplementary materials for RMW transitions)

$\tau \in \mathsf{Tid}$ (flush optimal markers). The persistence buffer P assigns a per-location persistence buffer to every non-volatile variable.[3]

We denote by $M.\dot{\mathtt{m}}$, $M.\tilde{\mathtt{m}}$, and $M.\mathtt{P}$ the components of a state $M \in \mathsf{PSC}.\mathtt{Q}$. We also write $M[\mathtt{X} \mapsto Y]$ for the state obtained from M by setting $M.\mathtt{X}$ to Y.

- The initial state is $M_{\mathsf{Init}} \stackrel{\mathsf{def}}{=} \langle \dot{m}_{\mathsf{Init}}, \tilde{m}_{\mathsf{Init}}, P_{\mathsf{Init}} \rangle$, where $\dot{m}_{\mathsf{Init}} \stackrel{\mathsf{def}}{=} \lambda \dot{x}.\, 0$, $\tilde{m}_{\mathsf{Init}} \stackrel{\mathsf{def}}{=} \lambda \tilde{x}.\, 0$, and $P_{\mathsf{Init}} \stackrel{\mathsf{def}}{=} \lambda \dot{x}.\, \epsilon$.
- 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 entry $M.\mathtt{P}(x)$ or, when there is no such entry, $M.\dot{\mathtt{m}}(x)$.

The transitions of PSC follow the intuitive account in the beginning of this section. 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. We note that RMWs to non-volatile variables (including those arising from failed compare-and-swap operations) include an implicit sfence transition. PSC is mostly oblivious to the thread that takes the step, except for $\mathtt{FO}(\dot{x})$ and $\mathtt{SF}$ steps for which the identity of the thread taking the step is important.

4.1 Extending PSC for Library Abstraction

Following §2, to have a more useful library abstraction theorem, we extend PSC with localized sfences and persistence blocks. In this section, we present the modifications and extensions needed in PSC for supporting these constructs. When referring to PSC in the sequel we mean the following revised version.

Local store fences. Localized sfences are straightforwardly supported by the

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

following additional memory transition:

$$\textsc{local sfence}\ \frac{l = \texttt{LSF}(\dot{X}) \qquad \forall \dot{x} \in \dot{X}.\, \texttt{FO}(\tau) \notin M.\texttt{P}(\dot{x})}{M \xrightarrow{\tau, l}_{\mathsf{PSC}} M}$$

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). In particular, we have $M \xrightarrow{\tau, \mathsf{LSF(NVVar)}}_{\mathsf{PSC}} M$ iff $M \xrightarrow{\tau, \mathsf{SF}}_{\mathsf{PSC}} M$.

Persistence blocks. The extension with persistence blocks is more involved. For this matter, 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 w.r.t. PSC described above:

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 $\langle B(\tau)(\dot{x}), \{\dot{x}\} \cup \dot{X} \rangle \in M.\texttt{Bid}$ for some $\dot{X} \subseteq \mathsf{NVVar}$.
3. The initial state is given by $M_{\mathsf{Init}} \stackrel{\mathsf{def}}{=} \langle \dot{m}_{\mathsf{Init}}, \tilde{m}_{\mathsf{Init}}, P_{\mathsf{Init}}, B_{\mathsf{Init}}, \mathit{Bid}_{\mathsf{Init}} \rangle$, where $B_{\mathsf{Init}} \stackrel{\mathsf{def}}{=} \lambda\tau.\, \lambda\dot{x}.\, \bot$, and $\mathit{Bid}_{\mathsf{Init}} \stackrel{\mathsf{def}}{=} \emptyset$.
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:

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

   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).

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

   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, then that block should not be currently active by any thread ($\forall \dot{x}, \tau.\, M.\mathtt{B}(\tau)(\dot{x}) \neq j$) and no entries of that block should remain in the volatile buffers ($\forall \dot{x}.\, j{:}\mathtt{W}(\_) \notin P'(\dot{x})$)).

4.2 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:

$$\textsc{synchronized}\ \frac{\alpha \in (\mathsf{Tid} \times \mathsf{Lab}) \cup \{\lightning\} \qquad \overline{q} \xrightarrow{\alpha}_{Pr} \overline{q}' \qquad M \xrightarrow{\alpha}_{\mathsf{PSC}} M'}{\langle \overline{q}, M\rangle \xrightarrow{\alpha}_{Pr \bowtie \mathsf{PSC}} \langle \overline{q}', M'\rangle} \qquad \textsc{program-internal}\ \frac{\alpha \in \mathsf{Tid} \times \{\epsilon\} \qquad \overline{q} \xrightarrow{\alpha}_{Pr} \overline{q}'}{\langle \overline{q}, M\rangle \xrightarrow{\alpha}_{Pr \bowtie \mathsf{PSC}} \langle \overline{q}', M\rangle}$$

$$\textsc{memory-internal}\ \frac{\alpha = \mathtt{per} \qquad M \xrightarrow{\alpha}_{\mathsf{PSC}} M'}{\langle \overline{q}, M\rangle \xrightarrow{\alpha}_{Pr \bowtie \mathsf{PSC}} \langle \overline{q}, M'\rangle}$$

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.

4.3 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 precisely capture each memory transition in terms of its effect on the two parts. In this section, we formalize this separation 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 relative to a set of variables. For persistence blocks to behave correctly, we need the following technical condition on this set: we say that a set $\dot{X} \subseteq \mathsf{NVVar}$ *separates a state* $M \in \mathsf{PSC.Q}$ if for every $\langle j, \dot{Y} \rangle \in M.\mathtt{Bid}$, we have $\dot{Y} \subseteq \dot{X}$ or $\dot{Y} \subseteq \mathsf{NVVar} \setminus \dot{X}$.

Definition 6. The *restriction* of $M \in \mathsf{PSC.Q}$ onto a set $X \subseteq \mathsf{Var}$ such that $X \cap \mathsf{NVVar}$ separates M, denoted by $M|_X$, is the state $M' \in \mathsf{PSC.Q}$ given by:
- $M'.\dot{\mathtt{m}}(\dot{x})$ is $M.\dot{\mathtt{m}}(\dot{x})$, if $\dot{x} \in \mathsf{NVVar} \cap X$, or 0 otherwise.
- $M'.\tilde{\mathtt{m}}(\tilde{x})$ is $M.\tilde{\mathtt{m}}(\tilde{x})$, 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, as well as transitions that induce a store fence (arising from RMWs to non-volatile variables), 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 $l \in \mathsf{Lab} \setminus \mathsf{SFLab}$ 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 $\tau \in \mathsf{Tid}$ and $l \in \mathsf{SFLab}$ which is either $\mathtt{SF}$ or has $\mathtt{var}(l) \in X$,
   $M_1 \xrightarrow{\tau,l}_{\mathsf{PSC}} M_2 \iff (M_1|_X \xrightarrow{\tau,l}_{\mathsf{PSC}} M_2|_X \wedge M_1|_{\mathsf{Var}\setminus X} \xrightarrow{\tau,\mathtt{SF}}_{\mathsf{PSC}} M_2|_{\mathsf{Var}\setminus X})$
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. $M_1 \xrightarrow{\lightning}_{\mathsf{PSC}} M_2 \iff (M_1|_X \xrightarrow{\lightning}_{\mathsf{PSC}} M_2|_X \wedge M_1|_{\mathsf{Var}\setminus X} \xrightarrow{\lightning}_{\mathsf{PSC}} M_2|_{\mathsf{Var}\setminus X})$

where $\mathsf{SFLab} \stackrel{\text{def}}{=} \{\mathtt{SF}\} \cup \{l \in \mathsf{Lab} \mid \mathtt{typ}(l) \in \{\mathtt{RMW}, \mathtt{R\text{-}ex}\} \wedge \mathtt{var}(l) \in \mathsf{NVVar}\}$.

The proof of Lemma 1 proceeds by standard case analysis ranging over all possible transitions of PSC.

Definition 7. 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 $M \in \mathsf{PSC.Q}$ defined by:

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

5 Libraries and Their Clients

In this section we present the notions of a library and a client that is using a particular library. Then, we define the necessary definitions for stating and proving the library abstraction theorem: histories and most general clients.

Libraries. We take a library L to be a function assigning a flat instruction sequence to method names in $dom(L) \subseteq \mathsf{F}$ (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 $\texttt{beginPB}(\dot{X})$).
- 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 8. 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:

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

$$\mathsf{HTLab}_F \stackrel{\text{def}}{=} (\mathsf{Tid} \times \mathsf{Lab}_F) \cup \{\lightning\}$$

Definition 9. 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 sequence over HTLab_F consisting of the following transition labels (in the same order they appear in t):

- call and return labels, $\langle \tau, \texttt{CALL}(f, \phi) \rangle$ and $\langle \tau, \texttt{RET}(f, \phi) \rangle$ with $f \in F$;
- crash labels; and
- an $\texttt{SF}$-label $\langle \tau, \texttt{SF} \rangle$ for every store-fence inducing label ($\langle \tau, l \rangle$ with $l \in \mathsf{SFLab}$).

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 a program 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 10. 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 libraries for persistent objects include a recovery method meant to be executed after a crash before any other library 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 a main method to each thread τ:

$MGC_{\mathsf{free}}(\tau)(\mathsf{main}) =$
`BEGIN` : `havoc` ;
`goto` $\mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n \mid$ `END` ;
$\mathtt{f}_1$: `call`(f_1) ; `goto BEGIN` ;
...
$\mathtt{f}_n$: `call`(f_n) ; `goto BEGIN` ;
`END` :

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

In MGC_{rec}, using CAS, one thread is selected to perform 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 11. 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 $\mathsf{H}(MGC[L]) \subseteq \mathsf{H}(MGC[L^{\#}])$.

Note that the library-correctness criterion, $L \sqsubseteq_{MGC} L^{\#}$, is necessary for contextual refinement to hold (otherwise, MGC itself is a client that can observe behaviors of L that are impossible for $L^{\#}$).

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 histories generated by the program, as well as with reachable states of the composition of the program Pr and the memory system PSC. The latter 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). Let libraries L and $L^{\#}$ and programs MGC and Pr be such that both L and $L^{\#}$ are safe for MGC and Pr, $L \sqsubseteq_{MGC} L^{\#}$ holds, and Pr correctly calls $L^{\#}$ w.r.t. MGC. Then, if $\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t}_{Pr[L] \bowtie \mathsf{PSC}} \langle \overline{q}, M \rangle$, 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 $\overline{q}^{\#}(\tau) = \overline{q}(\tau)$.
- $M^{\#}|_{\mathsf{Var}(Pr \setminus dom(L))} = M|_{\mathsf{Var}(Pr \setminus dom(L))}$.

Following the discussion in §2.3, we note that policy adherence is required to hold w.r.t. to $L^{\#}$. To prove the abstraction theorem, we use the following key lemma (in fact, it is used multiple times in the proof of Thm. 1 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 libraries L and L' implementing the same set F of methods be 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 $\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t}_{Pr[L] \bowtie \mathsf{PSC}} \langle \overline{q}, M \rangle$, where:

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

The proof of Lemma 2 (provided in the supplementary material) 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 are able to show that an analogous transition would be 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 are able to show that an analogous transition would be 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'. These client-local and library-local transition properties are possible to extract from t_{cl} and t_{lib} with the help of 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 and give a recipe for composing transition properties. For instance, transitions with action labels from $\mathsf{Lab} \setminus \mathsf{SFLab}$ can be composed, provided that the client program preserves the library memory state, and vice versa; while crashes and transitions with labels from SFLab record an interaction between a client program and a library and therefore need to be performed in synchrony. We extend these principles to ϵ-transitions, calls, and returns that do not affect the memory.

We use these two ideas in proving the Composition Lemma by induction on the sum of lengths of t_{cl} and t_{lib}. 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 $t_{\mathsf{cl}} = t'_{\mathsf{cl}} \cdot \alpha_{\mathsf{cl}}$ and $t_{\mathsf{lib}} = t'_{\mathsf{lib}} \cdot \alpha_{\mathsf{lib}}$, if the labels α_{cl} and α_{lib} both contribute the same history action to $\mathsf{H}_F(t_{\mathsf{cl}})$ and $\mathsf{H}_F(t_{\mathsf{lib}})$, α_{cl} and α_{lib} might be either different, if one of them is an RMW label, or equal otherwise. We use the local transition properties of α_{cl} and α_{lib} to compose them in synchrony. We use t' from the induction hypothesis for t'_{cl} and t'_{lib}, and let $t = t' \cdot \alpha_{\mathsf{cl}}$ or $t = t' \cdot \alpha_{\mathsf{lib}}$. If one of the labels α_{cl} and α_{lib} does not contribute a history action, for instance, α_{cl}, we use that the local transition property of α_{cl} preserves local transition properties of lib. We use t' from the induction hypothesis for t'_{cl} and t_{lib}, and let $t = t' \cdot \alpha_{\mathsf{cl}}$.

Using Lemma 2, 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 the classical notion of linearizability, our library-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, we define the composition of libraries $L_1, \ldots, L_n$ with pairwise disjoint sets of declared methods, denoted by $L_1 \uplus \ldots \uplus L_n$, to be the library obtained by taking the union of $L_1, \ldots, L_n$. Then, 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 is useful for establishing the library correctness condition $L \sqsubseteq_{MGC} L^\#$. Such conditions are typically established using simulation arguments with the observable transitions being those that induce history labels. The lemma below requires one to additionally identify a relation on non-volatile memories generated by $MGC[L] \bowtie \mathsf{PSC}$ and $MGC[L^\#] \bowtie \mathsf{PSC}$, show that it holds for the very initial memory, and that it is preserved during crashless executions assuming it holds initially. Thus, one can establish the library correctness condition by applying standard simulation arguments extended to relate the non-volatile memories for *crashless* traces.

Lemma 3. A trace t of $Pr \bowtie \mathsf{PSC}$ is called $\dot{m}_0$*-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 we have a binary relation R on $\mathsf{NVVar} \to \mathsf{Val}$ such that:

- $\langle \dot{m}_{\mathsf{Init}}, \dot{m}_{\mathsf{Init}} \rangle \in R$.
- 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 $L^\#$ has no $\mathtt{fo}(\cdot)$ and `sfence` instructions, then using the non-deterministic sfence steps (see §3), $MGC[L] \bowtie \mathsf{PSC}$ can take $\mathtt{SF}$-steps when $MGC[L^\#] \bowtie \mathsf{PSC}$ does, so store fences can be ignored when checking $\mathsf{H}(t) = \mathsf{H}(t^\#)$.

7 An Application: Persistent Pairs

In this section, we illustrate the use of the library abstraction theorem for a simple concurrent and persistent data structure, which is a pair of values that supports write and read operations. We present two specifications and an implementation for each specification. The two specifications ensure atomicity (i.e., linearizability if the system does not crash), and "data consistency" (reads always return two values written by a single invocation of write), but they differ in their exact persistency guarantees. For the concurrency aspect of the data structure, the implementations follow the sequence lock (seqlock, for short) mechanism, which uses a version counter along with the pair and allows readers to never block writers [6]. For durability, the implementations employ different techniques: one uses a "redo log" and the other is based on "checkpoints".

7.1 A Durable Pair

The first specification of the pair, 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 precise specification is as follows.

```
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 ;
```

The specification code uses a lock ($\tilde{\mathtt{l}}$) to ensure the atomicity of the data structure. For durability, writes use persistence blocks, which ensure that the two parts of the pair persist simultaneously. After the block is ended, $\mathtt{fl}(\dot{\mathtt{x}}_1)$ (which is 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 has completed, the written values are guaranteed to survive the crash. Thus, there is nothing to be done at recovery and the specification of recovery is a no-op. Nevertheless, aiming to allow implementations, the library policy requires that recovery is executed after every crash before any other method is invoked (as expressed by MGC_{rec} in §5).

Note that our simplified language has no mechanism for argument passing to/from methods. The specification above assumes that write receives arguments (read returns results) via designated registers, $\mathtt{a}_1$ and $\mathtt{a}_2$.

Next, we present an implementation of $L^{\#}_{\text{pair}}$, which we denote by L_{pair}. For clarity of presentation, we write $x := y$ instead of a read of y (to some fresh register) followed by a write to x. We also omit some 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) ;
lsfence(ẋ1^new, ẋ2^new) ;
ṡ := ṡ + 1 ; fl(ṡ) ;
ẋ1 := a1 ; fo(ẋ1) ; ẋ2 := a2 ; fo(ẋ2) ;
lsfence(ẋ1, ẋ2) ;
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) ;
lsfence(ẋ1, ẋ2) ;
END : ṡ := 0 ;
return ;
```

Ignoring crashes, atomicity is guaranteed here using the seqlock mechanism. We outline the key ideas behind the persistency management in this implementation. First, we observe 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 the write was in the middle of updating $\langle \dot{\mathtt{x}}_1, \dot{\mathtt{x}}_2 \rangle$ (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 a local store barrier when there is more than one variable to persist) is necessary here in order to restrict the out-of-order persistence and ensure the correctness of this implementation. The final write to $\dot{\mathtt{s}}$ 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. $L_{\text{pair}} \sqsubseteq_{MGC_{\mathsf{rec}}} L^{\#}_{\text{pair}}$.

A proof sketch is given in the supplementary material. It uses Lemma 3, letting $\langle \dot{m}, \dot{m}^{\#} \rangle \in R$ if the following hold:

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

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

7.2 A Buffered Durable Pair

An alternative specification of a pair, a library we denote by $L^{\#}_{\text{bpair}}$, allows for "buffered" behaviors that, following [22], aim to enable faster implementations by weaking persistency guarantees: instead of requiring operations to persist before returning to their caller, 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 $\mathtt{fl}(\dot{\mathtt{x}}_1)$ 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. Persistency is controlled by an additional method, called sync, that ensures that previous writes have reached persistent memory. Interestingly, without the sync method, an implementation could simply ignore persistency and store the pair in the volatile memory. Indeed, this corresponds to an execution of $L^{\#}_{\text{bpair}}$ in which the persistency buffers are never being flushed.

The implementation proposed below for this object exploits the freedom allowed by the specification. Writes and reads again follow the standard seqlock mechanism, 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.

```
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) ;
lsfence(ẋ1^prev, ẋ2^prev) ;
ḟ := 1 ; fl(ḟ) ;
NEXT : ẋ1^next := a1 ; fo(ẋ1^next) ;
ẋ2^next := a2 ; fo(ẋ2^next) ;
lsfence(ẋ1^next, ẋ2^next) ;
ḟ := 0 ; fl(ḟ) ;
UNLOCK : l̃ := 0 ;
return ;
```

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

A proof sketch is given in the supplementary material. It 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, [11] formalized the intuition that standard linearizability as defined in [19] 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 [14], resource-transferring programs [15], 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, 26].

Linearizability notions for persistent objects. Different approaches for adapting the standard linearizability criterion that is based on crash-free sequential specifications [19] were proposed before [3,17,22], but were not formally related to contextual refinement. Since methods like recover and sync (see §7.2) are meaningless in crash-free sequential specifications, they require a special external treatment in these linearizability adaptations. We believe that the variety of approaches to interpret crash-free sequential specifications for crash-resilient concurrent objects makes these notions hard to combine and apply.

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 §7.1), 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 NVM with full system crashes.

Durable linearizability [22] 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. Our language can express that, if every update method first records its task in a work-set, removes the task from the work-set, flushes the updated work-set, and performs 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 §7.1), 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` $\langle \mathtt{a}_1, \mathtt{a}_2 \rangle \in \dot{w}$;
`fl`($\dot{w}$) ;	remove $\langle \mathtt{a}_1, \mathtt{a}_2 \rangle$ from $\dot{w}$;
`UNLOCK1` : release $\tilde{\mathtt{l}}_{\mathtt{rw}}$;	`fl`($\dot{w}$) ;
... rest of the code as in write of $L^{\#}_{\text{pair}}$ (§7.1) ...	... write $\langle \mathtt{a}_1, \mathtt{a}_2 \rangle$ to $\langle \mathtt{x}, \mathtt{y} \rangle$ as in write of $L^{\#}_{\text{pair}}$ (§7.1) ...
recover :	`UNLOCK1` : release $\tilde{\mathtt{l}}_{\mathtt{rw}}$;
`return` ;	`BEGIN` : ... rest of the code as in read of $L^{\#}_{\text{pair}}$ (§7.1) ...

An alternative operational characterization of durable linearizability using Input/Output automata was developed in [10] and used to formally establish this property for the persistent queue of [12] by providing a full-blown simulation proof using the KIV proof assistant[4] Nevertheless, this work does not relate the proved correctness criterion to contextual refinement.

Persistency models. The underlying model we assume is PSC by [23], a strengthening of Px86 [28] that formalizes the Intel-x86 persistency.The paper [23] 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 [28] and ARM [9]; handling random access shared memory with allocations and deallocations (instead

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

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 [15] 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 [18,25], and formal logics as the one in [24]. Similarly, developing formal methods and tools that allow using library specifications for client reasoning it is left for future work, including decidable reachability analysis [2], program logics [27], and principled testing [13].

References

1. C++ reference. Accessed July-2021.
2. P. A. Abdulla, F. Haziza, L. Holík, B. Jonsson, and A. Rezine. An integrated specification and verification technique for highly concurrent data structures. In *TACAS'13*, pages 324–338, 2013.
3. M. K. Aguilera and S. Frølund. 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. Available at `https://developer.arm.com/documentation/ddi0487/latest/` [Online; accessed July-2021].
5. M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency. In *POPL*, pages 235–248, 2013.
6. H.-J. Boehm. Can Seqlocks get along with programming language memory models? In *MSPC*, pages 12–20, New York, NY, USA, 2012. ACM.
7. A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. Tractable refinement checking for concurrent objects. In *POPL*, page 651–662, New York, NY, USA, 2015. ACM.
8. S. Burckhardt, A. Gotsman, M. Musuvathi, and H. Yang. Concurrent library correctness on the tso memory model. In H. Seidl, editor, *Programming Languages and Systems*, pages 87–107, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
9. K. Cho, S.-H. Lee, A. Raad, and J. Kang. Revamping hardware persistency models: View-based and axiomatic persistency models for intel-x86 and armv8. In *PLDI*, PLDI 2021, page 16–31, New York, NY, USA, 2021. ACM.
10. J. Derrick, S. Doherty, B. Dongol, G. Schellhorn, and H. Wehrheim. Verifying correctness of persistent concurrent data structures: a sound and complete method. *Formal Aspects of Computing*, pages 1–27, 2021.
11. I. Filipović, P. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theoretical Computer Science*, 411(51):4379–4398, 2010.
12. M. Friedman, M. Herlihy, V. Marathe, and E. Petrank. A persistent lock-free queue for non-volatile memory. In *PPoPP*, pages 28–40, New York, NY, USA, 2018. ACM.
13. H. Gorjiara, G. H. Xu, and B. Demsky. Jaaru: Efficiently model checking persistent memory programs. In *ASPLOS*, page 415–428, New York, NY, USA, 2021. ACM.
14. A. Gotsman and H. Yang. Liveness-preserving atomicity abstraction. In L. Aceto, M. Henzinger, and J. Sgall, editors, *Automata, Languages and Programming*, pages 453–465, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

15. A. Gotsman and H. Yang. Linearizability with Ownership Transfer. *Logical Methods in Computer Science*, Volume 9, Issue 3, Sept. 2013.
16. R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. N. Wu, S.-C. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. In *POPL*, page 595–608, New York, NY, USA, 2015. ACM.
17. R. Guerraoui and R. R. Levy. Robust emulations of shared memory in a crash-recovery model. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, ICDCS '04, page 400–407, USA, 2004. IEEE Computer Society.
18. C. Hawblitzel, E. Petrank, S. Qadeer, and S. Tasiran. Automated and modular refinement reasoning for concurrent programs. In *CAV*, pages 449–465, Cham, 2015. Springer International Publishing.
19. M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.
20. Intel. Persistent Memory Programming, 2015.
21. Intel. Intel 64 and ia-32 architectures software developer's manual (combined volumes), May 2019. Order Number: 325462-069US.
22. J. Izraelevitz, H. Mendes, and M. L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *DISC*, pages 313–327, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.
23. A. Khyzha and O. Lahav. Taming x86-tso persistency. *Proc. ACM Program. Lang.*, 5(POPL), Jan. 2021.
24. H. Liang, X. Feng, and M. Fu. Rely-guarantee-based simulation for compositional verification of concurrent program transformations. *ACM Trans. Program. Lang. Syst.*, 36(1), Mar. 2014.
25. J. R. Lorch, Y. Chen, M. Kapritsos, B. Parno, S. Qadeer, U. Sharma, J. R. Wilcox, and X. Zhao. Armada: Low-effort verification of high-performance concurrent programs. In *PLDI*, page 197–210, New York, NY, USA, 2020. ACM.
26. A. Raad, M. Doko, L. Rožić, O. Lahav, and V. Vafeiadis. 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.
27. A. Raad, O. Lahav, and V. Vafeiadis. Persistent owicki-gries reasoning: A program logic for reasoning about persistent programs on intel-x86. *Proc. ACM Program. Lang.*, 4(OOPSLA), Nov. 2020.
28. A. Raad, J. Wickerson, G. Neiger, and V. Vafeiadis. Persistency semantics of the Intel-x86 architecture. *Proc. ACM Program. Lang.*, 4(POPL), Jan. 2020.
29. Y. Zuriel, M. Friedman, G. Sheffi, N. Cohen, and E. Petrank. Efficient lock-free durable sets. *Proc. ACM Program. Lang.*, 3(OOPSLA):128:1–128:26, Oct. 2019.

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

$$\frac{I(pc) = x := e \quad l = \texttt{W}(x, \phi(e))}{\langle pc, \phi\rangle \xrightarrow{l}_I \langle pc+1, \phi\rangle} \qquad \frac{I(pc) = r := x \quad l = \texttt{R}(x, v) \quad \phi' = \phi[r \mapsto v]}{\langle pc, \phi\rangle \xrightarrow{l}_I \langle pc+1, \phi'\rangle} \qquad \frac{I(pc) = r := \texttt{FADD}(x, e) \quad l = \texttt{RMW}(x, v, v + \phi(e)) \quad \phi' = \phi[r \mapsto v]}{\langle pc, \phi\rangle \xrightarrow{l}_I \langle pc+1, \phi'\rangle}$$

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

Fig. 2. Transitions of LTS induced by an instruction sequence

A Additional material for Section 3.2 (LTSs induced by instruction sequences)

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 given in Fig. 2.

B Auxiliary definitions for Section 4 (all transitions of PSC)

We use the following auxiliary function for looking up the most recent value of a variable:

$$M(x) \stackrel{\text{def}}{=} \begin{cases} v & x \in \mathsf{NVVar} \text{ and last write entry in } M.\mathtt{P}(x) \text{ has value } v \\ M.\dot{\mathtt{m}}(x) & x \in \mathsf{NVVar} \text{ and there are no write entries in } M.\mathtt{P}(x) \\ M.\tilde{\mathtt{m}}(x) & x \in \mathsf{VVar} \end{cases}$$

That is, when thread τ reads from a shared location x it obtains the latest accessible value of $\dot{x}$, which is defined by applying the following Mem function on the current persistent memory $\dot{m}$, the current persistence buffer P, and the location $\dot{x}$.

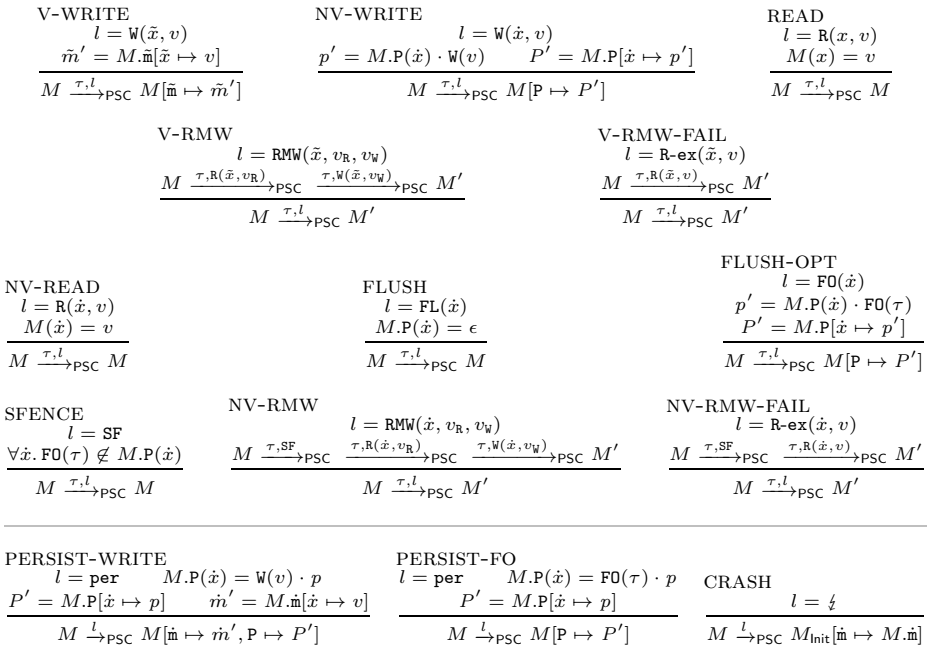

Fig. 3. Transitions of PSC

C Proofs

The following propositions are used in the following proofs. The all easily follow from our definitions.

Proposition 1. *If $h \in \mathsf{H}_F(Pr)$, then $h' \in \mathsf{H}_F(Pr)$ for every prefix h' of h.*

Proposition 2. *If $h \in \mathsf{H}_F(Pr)$, then $h \cdot \lightning \in \mathsf{H}_F(Pr)$.*

Proposition 3. *If $h \in \mathsf{H}_F(Pr)$, then $h \cdot \langle \tau, \mathtt{SF} \rangle \in \mathsf{H}_F(Pr)$ for every $\tau \in \mathsf{Tid}$.*

Proposition 4. *Suppose that $\langle \overline{q}, M \rangle \xrightarrow{t_1}_{Pr \bowtie \mathsf{PSC}} \langle \overline{q}_1, M_1 \rangle$ and $\langle \overline{q}, M \rangle \xrightarrow{t_2}_{Pr' \bowtie \mathsf{PSC}} \langle \overline{q}_2, M_2 \rangle$. If $\mathsf{H}_F(t_1) = \mathsf{H}_F(t_2)$, then for every τ we have $\overline{q}_1(\tau).\mathtt{f} \in F \iff \overline{q}_2(\tau).\mathtt{f} \in F$.*

The following properties all assume a library L that is safe for a program Pr.

Proposition 5. *If $\overline{q} \xrightarrow{\tau, l_\epsilon}_{Pr[L]} \overline{q}'$ and $\overline{q}(\tau).\mathtt{f} \notin dom(L)$, then $\overline{q} \xrightarrow{\tau, l_\epsilon}_{Pr} \overline{q}'$.*

Proposition 6. *For every state $\langle \overline{q}, M \rangle$ reachable in $Pr[L] \bowtie \mathsf{PSC}$, we have that both $\mathsf{Var}(L) \cap \mathsf{NVVar}$ and $\mathsf{Var}(Pr \setminus dom(L)) \cap \mathsf{NVVar}$ separate M.*

Proposition 7. *The following hold whenever $\overline{q} \xrightarrow{\tau, l}_{Pr[L]} \overline{q}'$:*
- *If $\overline{q}(\tau).\mathtt{f} \in dom(L)$, then $\mathtt{varset}(l) \subseteq \mathsf{Var}(L)$.*
- *If $\overline{q}(\tau).\mathtt{f} \notin dom(L)$, then $\mathtt{varset}(l) \subseteq \mathsf{Var}(Pr \setminus dom(L))$.*

The following propositions easily follow from the definitions in §4.

Proposition 8. *A set $\dot{X} \subseteq \mathsf{NVVar}$ separates M iff $\mathsf{NVVar} \setminus \dot{X}$ separates M.*

Proposition 9. *If $\dot{X} \subseteq \mathsf{NVVar}$ separates M_1 and $M_1 \xrightarrow{\alpha}_{\mathsf{PSC}} M_2$ with $\mathtt{varset}(\alpha) \subseteq \dot{X}$, then $\dot{X}$ separates M_2.*

Under the conditions of Def. 7, we always have the following properties:

Lemma 4. *Suppose $X_1, X_2 \subseteq \mathsf{Var}$ is such that $X_1 \cap X_2 = \emptyset$. Then*
(a) $\langle M_1, X_1 \rangle \uplus \langle M_2, X_2 \rangle = \langle M_2, X_2 \rangle \uplus \langle M_1, X_1 \rangle$.
(b) $\langle M_1, X_1 \rangle \uplus \langle M_2, X_2 \rangle = \langle M_1|_{X_1}, X_1 \rangle \uplus \langle M_2, X_2 \rangle$.
(c) $(\langle M_1, X_1 \rangle \uplus \langle M_2, X_2 \rangle)|_Y = M_1|_{X_1}$, *for any Y such that $X_1 \subseteq Y \subseteq \mathsf{Var} \setminus X_2$.*

Lemma 2 (Composition). Let libraries L and L' implementing the same set F of methods be 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 $\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t}_{Pr[L] \bowtie \mathsf{PSC}} \langle \overline{q}, M \rangle$, where:
- $\overline{q} = \lambda\tau. \begin{cases} \langle \overline{q}_{\mathsf{lib}}(\tau).\mathtt{pc}, \overline{q}_{\mathsf{lib}}(\tau).\phi, \overline{q}_{\mathsf{cl}}(\tau).\mathtt{pc_s}, \overline{q}_{\mathsf{cl}}(\tau).\mathtt{f} \rangle & \overline{q}_{\mathsf{cl}}(\tau).\mathtt{f} \in F \\ \overline{q}_{\mathsf{cl}}(\tau) & \text{otherwise} \end{cases}$
- $M = \langle M_{\mathsf{cl}}|_{\mathsf{Var}(Pr \setminus F)}, \mathsf{Var}(Pr \setminus F) \rangle \uplus \langle M_{\mathsf{lib}}|_{\mathsf{Var}(L)}, \mathsf{Var}(L) \rangle$

Proof. Consider two libraries L and L' implementing the same method set F, both safe for a program Pr, with L being also safe for a program Pr'. For traces t_{cl} and t_{lib}, let $\textsc{Compose}(t_{\mathsf{cl}}, t_{\mathsf{lib}})$ denote the rest of the statement of the lemma. We prove $(\forall t_{\mathsf{cl}}, t_{\mathsf{lib}}.\, \textsc{Compose}(t_{\mathsf{cl}}, t_{\mathsf{lib}}))$ by induction on the sum of lengths of t_{cl} and t_{lib}.

The base of induction is to show $\textsc{Compose}(t_{\mathsf{cl}}, t_{\mathsf{lib}})$ when $|t_{\mathsf{cl}}| + |t_{\mathsf{lib}}| = 0$; then $\langle \overline{q}, M\rangle = \langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}}\rangle$, and we can simply take $t = \epsilon$.

The step of induction is to show $\textsc{Compose}(t_{\mathsf{cl}}, t_{\mathsf{lib}})$, assuming that $\textsc{Compose}(t'_{\mathsf{cl}}, t'_{\mathsf{lib}})$ holds for every t'_{cl} and t'_{lib} with $|t'_{\mathsf{cl}}| + |t'_{\mathsf{lib}}| < |t_{\mathsf{cl}}| + |t_{\mathsf{lib}}|$. We split the rest of the proof into the following cases:

(I) t_{lib} is non-empty and ends with a label α_{lib} that does not contribute to $\mathsf{H}_F(t_{\mathsf{lib}})$, i.e., one of the following holds:
- $\alpha_{\mathsf{lib}} = \mathtt{per}$
- $\alpha_{\mathsf{lib}} \in \mathsf{Tid} \times \{\epsilon\}$
- $\alpha_{\mathsf{lib}} \in \mathsf{Tid} \times \{\mathtt{CALL}(f, \phi), \mathtt{RET}(f, \phi) \in \mathsf{Lab} \mid f \notin F\}$
- $\alpha_{\mathsf{lib}} \in \mathsf{Tid} \times \{l \in \mathsf{Lab} \mid \mathtt{typ}(l) \notin \{\mathtt{CALL}, \mathtt{RET}\} \wedge l \notin \mathsf{SFLab}\}$

(II) t_{cl} is non-empty and ends with a label α_{cl} that does not contribute to $\mathsf{H}_F(t_{\mathsf{cl}})$;
- $\alpha_{\mathsf{cl}} = \mathtt{per}$
- $\alpha_{\mathsf{cl}} \in \mathsf{Tid} \times \{\epsilon\}$
- $\alpha_{\mathsf{cl}} \in \mathsf{Tid} \times \{\mathtt{CALL}(f, \phi), \mathtt{RET}(f, \phi) \in \mathsf{Lab} \mid f \notin F\}$
- $\alpha_{\mathsf{cl}} \in \mathsf{Tid} \times \{l \in \mathsf{Lab} \mid \mathtt{typ}(l) \notin \{\mathtt{CALL}, \mathtt{RET}\} \wedge l \notin \mathsf{SFLab}\}$

(III) both t_{cl} and t_{lib} are non-empty and end with labels α_{cl} and α_{lib} contributing to histories $\mathsf{H}_F(t_{\mathsf{cl}})$ and $\mathsf{H}_F(t_{\mathsf{lib}})$, i.e., one of the following holds:
- $\alpha_{\mathsf{cl}} = \alpha_{\mathsf{lib}} \in \mathsf{Tid} \times \{\mathtt{CALL}(f, \phi) \in \mathsf{Lab} \mid f \in F\}$
- $\alpha_{\mathsf{cl}} = \alpha_{\mathsf{lib}} \in \mathsf{Tid} \times \{\mathtt{RET}(f, \phi) \in \mathsf{Lab} \mid f \in F\}$
- $\alpha_{\mathsf{cl}} = \alpha_{\mathsf{lib}} = \lightning$
- $\alpha_{\mathsf{cl}}, \alpha_{\mathsf{lib}} \in \mathsf{Tid} \times \mathsf{SFLab}$

It is easy to see that these three cases exhaust all possibilities for t_{lib} and t_{cl}. For instance, suppose that t_{lib} is non-empty, but ends with a label corresponding to a history label. Let $t_{\mathsf{lib}} = \_ \cdot \alpha_{\mathsf{lib}}$ and $\mathsf{H}_F(t_{\mathsf{lib}}) = \_ \cdot \mathsf{H}_F(\alpha_{\mathsf{lib}})$. By the lemma's premise, $\mathsf{H}_F(t_{\mathsf{cl}}) = \mathsf{H}_F(t_{\mathsf{lib}})$. Therefore, it must be that $t_{\mathsf{cl}} = \_ \cdot \alpha_{\mathsf{cl}} \cdot t'_{\mathsf{cl}}$ and $\mathsf{H}_F(t_{\mathsf{cl}}) = \_ \cdot \mathsf{H}_F(\alpha_{\mathsf{cl}})$. However, when t'_{cl} is non-empty, such a possibility is already covered by Case II, and when t'_{cl} is empty, such a possibility is already covered by Case III.

Case I. Suppose that t_{lib} is non-empty and ends with a label α_{lib} not corresponding to a history label. Let $t_{\mathsf{lib}} = t'_{\mathsf{lib}} \cdot \alpha_{\mathsf{lib}}$, and consider any state $\langle \overline{q}'_{\mathsf{lib}}, M'_{\mathsf{lib}}\rangle$ for which there are the following transitions:

$$\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}}\rangle \xrightarrow{t'_{\mathsf{lib}}}_{Pr'[L] \bowtie \mathsf{PSC}} \langle \overline{q}'_{\mathsf{lib}}, M'_{\mathsf{lib}}\rangle \xrightarrow{\alpha_{\mathsf{lib}}}_{Pr'[L] \bowtie \mathsf{PSC}} \langle \overline{q}_{\mathsf{lib}}, M_{\mathsf{lib}}\rangle \qquad \text{(I)}$$

In the following, we consider differently various cases for α_{lib} in order to construct t.

Suppose $\alpha_{\mathsf{lib}} = \mathtt{per}$. The last transition of Eq. (I) is memory-internal, so $\overline{q}'_{\mathsf{lib}} = \overline{q}_{\mathsf{lib}}$. Then $\overline{q} = \overline{q}'$ holds by construction. We deduce from Eq. (I) that $M'_{\mathsf{lib}} \xrightarrow{\mathtt{per}}_{\mathsf{PSC}} M_{\mathsf{lib}}$ holds. By Prop. 6, since L is safe for Pr', $\mathsf{Var}(L) \cap \mathsf{NVVar}$ separates M'_{lib}, which allows us to deduce $M'_{\mathsf{lib}}|_{\mathsf{Var}(L)} \xrightarrow{\mathtt{per}}_{\mathsf{PSC}} M_{\mathsf{lib}}|_{\mathsf{Var}(L)}$ by Lemma 1(3). From

the induction hypothesis $\textsc{Compose}(t_{\mathsf{cl}}, t'_{\mathsf{lib}})$, we know that $M' = \langle M_{\mathsf{cl}}|_{\mathsf{Var}(Pr \setminus F)}, \mathsf{Var}(Pr \setminus F)\rangle \uplus \langle M'_{\mathsf{lib}}|_{\mathsf{Var}(L)}, \mathsf{Var}(L)\rangle$, so overall we obtain $M'|_{\mathsf{Var}(L)} \xrightarrow{\mathtt{per}}_{\mathsf{PSC}} M|_{\mathsf{Var}(L)}$. Also, $M'_{\mathsf{cl}}|_{\mathsf{Var}(Pr\setminus F)} \xrightarrow{\mathtt{per}}_{\mathsf{PSC}} M'_{\mathsf{cl}}|_{\mathsf{Var}(Pr\setminus F)}$ holds trivially. Note that by Lemma 4(c), $M'|_{\mathsf{Var}\setminus\mathsf{Var}(L)} = M|_{\mathsf{Var}\setminus\mathsf{Var}(L)} = M_{\mathsf{cl}}|_{\mathsf{Var}(Pr\setminus F)}$. Therefore, we get $M'|_{\mathsf{Var}\setminus\mathsf{Var}(L)} \xrightarrow{\mathtt{per}}_{\mathsf{PSC}} M|_{\mathsf{Var}\setminus\mathsf{Var}(L)}$. Since $\langle \overline{q}', M'\rangle$ is reachable, by Prop. 6, $\mathsf{Var}(L) \cap \mathsf{NVVar}$ separates M'. Then, by Lemma 1(3), we obtain $M' \xrightarrow{\mathtt{per}}_{\mathsf{PSC}} M$. The transition is memory-internal, so we get $\langle \overline{q}', M'\rangle \xrightarrow{\mathtt{per}}_{Pr[L]\bowtie\mathsf{PSC}} \langle \overline{q}, M\rangle$,

By the induction hypothesis $\textsc{Compose}(t_{\mathsf{cl}}, t'_{\mathsf{lib}})$, for $\langle \overline{q}_{\mathsf{cl}}, M_{\mathsf{cl}}\rangle$, $\langle \overline{q}'_{\mathsf{lib}}, M'_{\mathsf{lib}}\rangle$ there there exists t' such that $\mathsf{H}(t') = \mathsf{H}(t_{\mathsf{cl}})$ and $\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}}\rangle \xrightarrow{t'}_{Pr[L]\bowtie\mathsf{PSC}} \langle \overline{q}', M'\rangle$. It is easy to see that $\mathsf{H}(t' \cdot \mathtt{per}) = \mathsf{H}(t_{\mathsf{cl}})$, and we have shown that:

$$\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}}\rangle \xrightarrow{t'}_{Pr[L]\bowtie\mathsf{PSC}} \langle \overline{q}', M'\rangle \xrightarrow{\mathtt{per}}_{Pr[L]\bowtie\mathsf{PSC}} \langle \overline{q}, M\rangle$$

To conclude the proof for this case, we let $t = t' \cdot \mathtt{per}$.

Suppose $\alpha_{\mathsf{lib}} = \langle \tau, \epsilon\rangle$. The transition is program-internal, so $M'_{\mathsf{lib}} = M_{\mathsf{lib}}$. Therefore, by construction, $M' = M$. The histories of t_{cl}, t'_{lib} and t_{lib} coincide, so there are two possibilities: either the execution in all of them is outside of a method of L (and then $\overline{q}_{\mathsf{cl}}(\tau).\mathtt{f}, \overline{q}'_{\mathsf{lib}}(\tau).\mathtt{f}, \overline{q}_{\mathsf{lib}}(\tau).\mathtt{f} \notin F$ holds) or inside of the same method (and then $\overline{q}_{\mathsf{cl}}(\tau).\mathtt{f}, \overline{q}'_{\mathsf{lib}}(\tau).\mathtt{f}, \overline{q}_{\mathsf{lib}}(\tau).\mathtt{f} \in F$ holds). We first assume that $\overline{q}_{\mathsf{cl}}(\tau).\mathtt{f}, \overline{q}'_{\mathsf{lib}}(\tau).\mathtt{f}, \overline{q}_{\mathsf{lib}}(\tau).\mathtt{f} \notin F$. Then $\overline{q}(\tau) = \overline{q}'(\tau) = \overline{q}_{\mathsf{cl}}(\tau)$ holds. We let $t = t'$; then the induction step immediately follows from the induction hypothesis $\textsc{Compose}(t_{\mathsf{cl}}, t'_{\mathsf{lib}})$. We now consider the possibility of $\overline{q}_{\mathsf{cl}}(\tau).\mathtt{f}, \overline{q}'_{\mathsf{lib}}(\tau).\mathtt{f}, \overline{q}_{\mathsf{lib}}(\tau).\mathtt{f} \in F$. From Eq. (I) we deduce that $\langle \overline{q}'_{\mathsf{lib}}(\tau).\mathtt{pc}, \overline{q}'_{\mathsf{lib}}(\tau).\phi\rangle \xrightarrow{\epsilon}_{L(\overline{q}_{\mathsf{lib}}(\tau).\mathtt{f})} \langle \overline{q}_{\mathsf{lib}}(\tau).\mathtt{pc}, \overline{q}_{\mathsf{lib}}(\tau).\phi\rangle$ holds. By construction of $\overline{q}$ and $\overline{q}'$, and the concurrent program transition rules we obtain that $\overline{q}' \xrightarrow{\tau,\epsilon}_{Pr} \overline{q}$ holds. Since the latter is a program-internal transition, we get $\langle \overline{q}', M'\rangle \xrightarrow{\tau,\epsilon}_{Pr[L]\bowtie\mathsf{PSC}} \langle \overline{q}, M\rangle$.

By the induction hypothesis $\textsc{Compose}(t_{\mathsf{cl}}, t'_{\mathsf{lib}})$, for $\langle \overline{q}_{\mathsf{cl}}, M_{\mathsf{cl}}\rangle$, $\langle \overline{q}'_{\mathsf{lib}}, M'_{\mathsf{lib}}\rangle$ there there exists t' such that $\mathsf{H}(t') = \mathsf{H}(t_{\mathsf{cl}})$ and $\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}}\rangle \xrightarrow{t'}_{Pr[L]\bowtie\mathsf{PSC}} \langle \overline{q}', M'\rangle$. It is easy to see that $\mathsf{H}(t' \cdot \langle \tau, \epsilon\rangle) = \mathsf{H}(t_{\mathsf{cl}})$, and we have shown that:

$$\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}}\rangle \xrightarrow{t'}_{Pr[L]\bowtie\mathsf{PSC}} \langle \overline{q}', M'\rangle \xrightarrow{\tau,\epsilon}_{Pr[L]\bowtie\mathsf{PSC}} \langle \overline{q}, M\rangle$$

To conclude the proof for this case, we let $t = t' \cdot \langle \tau, \epsilon\rangle$.

Suppose $\alpha_{\mathsf{lib}} = \langle \tau, \mathtt{CALL}(f, \phi)\rangle$ (or $\alpha_{\mathsf{lib}} = \langle \tau, \mathtt{RET}(f, \phi)\rangle$) and $f \notin F$. The transition is program-internal, so $M'_{\mathsf{lib}} = M_{\mathsf{lib}}$. Therefore, by construction, $M' = M$. It is also a call (or return) transition, so necessarily $\overline{q}'_{\mathsf{lib}}(\tau).\mathtt{f} = \mathsf{main} \notin F$ and $\overline{q}_{\mathsf{lib}}(\tau).\mathtt{f} = f \notin F$ (or $\overline{q}'_{\mathsf{lib}}(\tau).\mathtt{f} = f \notin F$ and $\overline{q}_{\mathsf{lib}}(\tau).\mathtt{f} = \mathsf{main} \notin F$). By the premise of the induction step, $\mathsf{H}_F(t_{\mathsf{cl}}) = \mathsf{H}_F(t_{\mathsf{lib}})$, so it can only be that $\overline{q}_{\mathsf{cl}}(\tau).\mathtt{f} \notin F$ holds. Then, by construction, $\overline{q}(\tau) = \overline{q}'(\tau) = \overline{q}_{\mathsf{cl}}(\tau)$ holds. We let $t = t'$; then the induction step immediately follows from the induction hypothesis $\textsc{Compose}(t_{\mathsf{cl}}, t'_{\mathsf{lib}})$.

Suppose $\alpha_{\mathsf{lib}} = \langle \tau, l\rangle$ and $\mathtt{typ}(l) \notin \{\mathtt{CALL}, \mathtt{RET}\} \wedge l \notin \mathsf{SFLab}$.

By the premise of the induction, $\mathsf{H}_F(t_{\mathsf{cl}}) = \mathsf{H}_F(t_{\mathsf{lib}})$. Also, $\mathsf{H}_F(t_{\mathsf{lib}}) = \mathsf{H}_F(t'_{\mathsf{lib}})$. Hence, either the execution in all of the traces is outside of a method of L (and then $\overline{q}_{\mathsf{cl}}(\tau).\mathtt{f}, \overline{q}'_{\mathsf{lib}}(\tau).\mathtt{f}, \overline{q}_{\mathsf{lib}}(\tau).\mathtt{f} \notin F$ holds) or inside of the same method (and then $\overline{q}_{\mathsf{cl}}(\tau).\mathtt{f}, \overline{q}'_{\mathsf{lib}}(\tau).\mathtt{f}, \overline{q}_{\mathsf{lib}}(\tau).\mathtt{f} \in F$ holds). We first assume that $\overline{q}_{\mathsf{cl}}(\tau).\mathtt{f}, \overline{q}'_{\mathsf{lib}}(\tau).\mathtt{f}, \overline{q}_{\mathsf{lib}}(\tau).\mathtt{f} \notin F$. Then $\overline{q} = \overline{q}' = \overline{q}_{\mathsf{cl}}$ holds. By Prop. 7, $\mathtt{varset}(l) \subseteq \mathsf{Var}\setminus\mathsf{Var}(L)$. From Eq. (I) we

deduce that $M'_{\mathsf{lib}} \xrightarrow{\tau,l}_{\mathsf{PSC}} M_{\mathsf{lib}}$ holds. Since $\langle \overline{q}'_{\mathsf{lib}}, M'_{\mathsf{lib}} \rangle$ is reachable, by Prop. 6, we have that $\mathsf{Var}(L)$ separates M'. By Lemma 1(1), $M'_{\mathsf{lib}}|_{\mathsf{Var}\setminus\mathsf{Var}(L)} \xrightarrow{\tau,l}_{\mathsf{PSC}} M_{\mathsf{lib}}|_{\mathsf{Var}\setminus\mathsf{Var}(L)}$ and $M'_{\mathsf{lib}}|_{\mathsf{Var}(L)} = M_{\mathsf{lib}}|_{\mathsf{Var}(L)}$. The latter in particular implies that, by construction, $M' = M$. We let $t = t'$; then the induction step immediately follows from the induction hypothesis $\textsc{Compose}(t_{\mathsf{cl}}, t'_{\mathsf{lib}})$.

We now consider the possibility of $\overline{q}_{\mathsf{cl}}(\tau).\mathtt{f}, \overline{q}'_{\mathsf{lib}}(\tau).\mathtt{f}, \overline{q}_{\mathsf{lib}}(\tau).\mathtt{f} \in F$. From Eq. (I) we deduce that $\langle \overline{q}'_{\mathsf{lib}}(\tau).\mathtt{pc}, \overline{q}'_{\mathsf{lib}}(\tau).\phi \rangle \xrightarrow{l}_{L(\overline{q}_{\mathsf{lib}}(\tau).\mathtt{f})} \langle \overline{q}_{\mathsf{lib}}(\tau).\mathtt{pc}, \overline{q}_{\mathsf{lib}}(\tau).\phi \rangle$ holds. By construction of $\overline{q}$ and $\overline{q}'$, and the concurrent program transition rules we obtain that $\overline{q}' \xrightarrow{\tau,l}_{Pr} \overline{q}$ holds. Secondly, by $\textsc{Compose}(t_{\mathsf{cl}}, t'_{\mathsf{lib}})$, $M' = \langle M_{\mathsf{cl}}|_{\mathsf{Var}(Pr\setminus F)}, \mathsf{Var}(Pr \setminus F) \rangle \uplus \langle M'_{\mathsf{lib}}|_{\mathsf{Var}(L)}, \mathsf{Var}(L) \rangle$. By Prop. 7, $\mathtt{varset}(l) \subseteq \mathsf{Var} \setminus \mathsf{Var}(L)$. We deduce from Eq. (I) $M'_{\mathsf{lib}} \xrightarrow{\tau,l}_{\mathsf{PSC}} M_{\mathsf{lib}}$. By Prop. 6, since L is safe for Pr', $\mathsf{Var}(L) \cap \mathsf{NVVar}$ separates M'_{lib}, which allows us to deduce $M'_{\mathsf{lib}}|_{\mathsf{Var}(L)} \xrightarrow{\tau,l}_{\mathsf{PSC}} M_{\mathsf{lib}}|_{\mathsf{Var}(L)}$ by Lemma 1(3). From the induction hypothesis $\textsc{Compose}(t_{\mathsf{cl}}, t'_{\mathsf{lib}})$, we know that $M' = \langle M_{\mathsf{cl}}|_{\mathsf{Var}(Pr\setminus F)}, \mathsf{Var}(Pr \setminus F) \rangle \uplus \langle M'_{\mathsf{lib}}|_{\mathsf{Var}(L)}, \mathsf{Var}(L) \rangle$, so overall we obtain $M'|_{\mathsf{Var}(L)} \xrightarrow{\tau,l}_{\mathsf{PSC}} M|_{\mathsf{Var}(L)}$. Also, note that by Lemma 4(c) and by contruction of the merges, $M'|_{\mathsf{Var}\setminus\mathsf{Var}(L)} = M|_{\mathsf{Var}\setminus\mathsf{Var}(L)} = M_{\mathsf{cl}}|_{\mathsf{Var}(Pr\setminus F)}$. Since $\langle \overline{q}', M' \rangle$ is reachable, by Prop. 6, we have that $\mathsf{Var}(L)$ separates M'. Then, by Lemma 1(1), we have $M' \xrightarrow{\tau,l}_{\mathsf{PSC}} M$. By synchronizing the latter with $\overline{q}' \xrightarrow{\tau,l}_{Pr} \overline{q}$, we get: $\langle \overline{q}', M' \rangle \xrightarrow{\tau,l}_{Pr[L]\bowtie\mathsf{PSC}} \langle \overline{q}, M \rangle$.

By the induction hypothesis $\textsc{Compose}(t_{\mathsf{cl}}, t'_{\mathsf{lib}})$, for $\langle \overline{q}_{\mathsf{cl}}, M_{\mathsf{cl}} \rangle$, $\langle \overline{q}'_{\mathsf{lib}}, M'_{\mathsf{lib}} \rangle$ there there exists t' such that $\mathsf{H}(t') = \mathsf{H}(t_{\mathsf{cl}})$ and $\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t'}_{Pr[L]\bowtie\mathsf{PSC}} \langle \overline{q}', M' \rangle$. It is easy to see that $\mathsf{H}(t' \cdot \langle \tau, l \rangle) = \mathsf{H}(t_{\mathsf{cl}})$, and we have shown that:

$$\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t'}_{Pr[L]\bowtie\mathsf{PSC}} \langle \overline{q}', M' \rangle \xrightarrow{\tau,l}_{Pr[L]\bowtie\mathsf{PSC}} \langle \overline{q}, M \rangle$$

To conclude the proof for this case, we let $t = t' \cdot \langle \tau, l \rangle$.

Case II. Suppose that t_{cl} is non-empty and ends with a label α_{cl} not corresponding to a history label. Let $t_{\mathsf{cl}} = t'_{\mathsf{cl}} \cdot \alpha_{\mathsf{cl}}$, and consider any state $\langle \overline{q}'_{\mathsf{cl}}, M'_{\mathsf{cl}} \rangle$ for which there are the following transitions:

$$\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t'_{\mathsf{cl}}}_{Pr[L']\bowtie\mathsf{PSC}} \langle \overline{q}'_{\mathsf{cl}}, M'_{\mathsf{cl}} \rangle \xrightarrow{\alpha_{\mathsf{cl}}}_{Pr[L']\bowtie\mathsf{PSC}} \langle \overline{q}_{\mathsf{cl}}, M_{\mathsf{cl}} \rangle \qquad \text{(II)}$$

Like in Case I, we consider separately various cases for α_{cl} in order to construct t. We only give a proof for the case of call and return labels here, since the other cases are analogous to Case I.

Suppose $\alpha_{\mathsf{cl}} = \langle \tau, \mathtt{CALL}(f, \phi) \rangle$ (or $\alpha_{\mathsf{cl}} = \langle \tau, \mathtt{RET}(f, \phi) \rangle$) and $f \notin F$. The transition is program-internal, so $M'_{\mathsf{cl}} = M_{\mathsf{cl}}$. It is also a call transition into a method not in F, so $\overline{q}'_{\mathsf{cl}}(\tau).\mathtt{f} = \mathsf{main} \notin F$ and $\overline{q}'_{\mathsf{cl}}(\tau).\mathtt{f} \notin F$. Then, by construction, $\overline{q}'(\tau) = \overline{q}'_{\mathsf{cl}}$ and $\overline{q}(\tau) = \overline{q}_{\mathsf{cl}}(\tau)$. We deduce from Eq. (II) that $\overline{q}'_{\mathsf{cl}} \xrightarrow{\tau,\mathtt{CALL}(f,\phi)}_{Pr} \overline{q}_{\mathsf{cl}}$ and, therefore, $\overline{q}' \xrightarrow{\tau,\mathtt{CALL}(f,\phi)}_{Pr} \overline{q}$. Since the transition is program-internal, $\langle \overline{q}', M' \rangle \xrightarrow{\tau,\mathtt{CALL}(f,\phi)}_{Pr[L]\bowtie\mathsf{PSC}} \langle \overline{q}, M \rangle$.

By the induction hypothesis $\textsc{Compose}(t'_{\mathsf{cl}}, t_{\mathsf{lib}})$, for $\langle \overline{q}'_{\mathsf{cl}}, M'_{\mathsf{cl}} \rangle$, $\langle \overline{q}_{\mathsf{lib}}, M_{\mathsf{lib}} \rangle$ there exists t' such that $\mathsf{H}(t') = \mathsf{H}(t'_{\mathsf{cl}})$ and $\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t'}_{Pr[L]\bowtie\mathsf{PSC}} \langle \overline{q}', M' \rangle$. It is easy to see that $\mathsf{H}(t' \cdot \langle \tau, \mathtt{CALL}(f, \phi) \rangle) = \mathsf{H}(t'_{\mathsf{cl}} \cdot \langle \tau, \mathtt{CALL}(f, \phi) \rangle) = \mathsf{H}(t_{\mathsf{cl}})$, and we have shown that:

$$\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t'}_{Pr[L]\bowtie\mathsf{PSC}} \langle \overline{q}', M' \rangle \xrightarrow{\tau,\mathtt{CALL}(f,\phi)}_{Pr[L]\bowtie\mathsf{PSC}} \langle \overline{q}, M \rangle$$

To conclude the proof for this case, we let $t = t' \cdot \langle \tau, \mathtt{CALL}(f, \phi) \rangle$.

Suppose $\alpha_{\mathsf{cl}} = \langle \tau, \mathtt{RET}(f, \phi) \rangle$ and $f \notin F$. The transition is program-internal, so $M'_{\mathsf{cl}} = M_{\mathsf{cl}}$. It is also a return transition from a method not in F, so $\overline{q}'_{\mathsf{cl}}(\tau).\mathtt{f} \notin F$ and $\overline{q}'_{\mathsf{cl}}(\tau).\mathtt{f} = \mathsf{main} \notin F$. Then, by construction, $\overline{q}'(\tau) = \overline{q}'_{\mathsf{cl}}$ and $\overline{q}(\tau) = \overline{q}_{\mathsf{cl}}(\tau)$. We deduce from Eq. (II) that $\overline{q}'_{\mathsf{cl}} \xrightarrow{\tau, \mathtt{RET}(f,\phi)}_{Pr} \overline{q}_{\mathsf{cl}}$ and, therefore, $\overline{q}' \xrightarrow{\tau, \mathtt{RET}(f,\phi)}_{Pr} \overline{q}$. Since the transition is program-internal, $\langle \overline{q}', M' \rangle \xrightarrow{\tau, \mathtt{RET}(f,\phi)}_{Pr[L] \bowtie \mathsf{PSC}} \langle \overline{q}, M \rangle$.

By the induction hypothesis $\textsc{Compose}(t'_{\mathsf{cl}}, t_{\mathsf{lib}})$, for $\langle \overline{q}'_{\mathsf{cl}}, M'_{\mathsf{cl}} \rangle$, $\langle \overline{q}_{\mathsf{lib}}, M_{\mathsf{lib}} \rangle$ there exists t' such that $\mathsf{H}(t') = \mathsf{H}(t'_{\mathsf{cl}})$ and $\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t'}_{Pr[L] \bowtie \mathsf{PSC}} \langle \overline{q}', M' \rangle$. It is easy to see that $\mathsf{H}(t' \cdot \langle \tau, \mathtt{RET}(f, \phi) \rangle) = \mathsf{H}(t'_{\mathsf{cl}} \cdot \langle \tau, \mathtt{RET}(f, \phi) \rangle) = \mathsf{H}(t_{\mathsf{cl}})$, and we have shown that:

$$\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t'}_{Pr[L] \bowtie \mathsf{PSC}} \langle \overline{q}', M' \rangle \xrightarrow{\tau, \mathtt{RET}(f,\phi)}_{Pr[L] \bowtie \mathsf{PSC}} \langle \overline{q}, M \rangle$$

To conclude the proof for this case, we let $t = t' \cdot \langle \tau, \mathtt{RET}(f, \phi) \rangle$.

Case III. Suppose both t_{cl} and t_{lib} are non-empty and end with a label corresponding to a history label. Let $t_{\mathsf{cl}} = t'_{\mathsf{cl}} \cdot \alpha_{\mathsf{cl}}$ and $t_{\mathsf{lib}} = t'_{\mathsf{lib}} \cdot \alpha_{\mathsf{lib}}$. By the premise of the induction step, $\mathsf{H}_F(t_{\mathsf{cl}}) = \mathsf{H}_F(t_{\mathsf{lib}})$ holds; hence, $\mathsf{H}_F(\alpha_{\mathsf{cl}}) = \mathsf{H}_F(\alpha_{\mathsf{lib}})$, and we refer to that history action label as α. Let $\langle \overline{q}'_{\mathsf{lib}}, M'_{\mathsf{lib}} \rangle$ and $\langle \overline{q}'_{\mathsf{cl}}, M'_{\mathsf{cl}} \rangle$ be any states for which there are the following transitions:

$$\begin{array}{c} \langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t'_{\mathsf{lib}}}_{Pr'[L] \bowtie \mathsf{PSC}} \langle \overline{q}'_{\mathsf{lib}}, M'_{\mathsf{lib}} \rangle \xrightarrow{\alpha_{\mathsf{lib}}}_{Pr'[L] \bowtie \mathsf{PSC}} \langle \overline{q}_{\mathsf{lib}}, M_{\mathsf{lib}} \rangle \\ \langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t'_{\mathsf{cl}}}_{Pr[L'] \bowtie \mathsf{PSC}} \langle \overline{q}'_{\mathsf{cl}}, M'_{\mathsf{cl}} \rangle \xrightarrow{\alpha_{\mathsf{cl}}}_{Pr[L'] \bowtie \mathsf{PSC}} \langle \overline{q}_{\mathsf{cl}}, M_{\mathsf{cl}} \rangle \end{array} \tag{III}$$

In the following, we consider different combinations of α_{cl} and α_{lib} in order to construct t.

Suppose $\alpha = \alpha_{\mathsf{cl}} = \alpha_{\mathsf{lib}} = \langle \tau, \mathtt{CALL}(f, \phi) \rangle$. Let $\overline{q}'_{\mathsf{cl}}(\tau).\mathtt{pc} = pc$. The transition is program-internal, so $M'_{\mathsf{lib}} = M_{\mathsf{lib}}$ and $M'_{\mathsf{cl}} = M_{\mathsf{cl}}$; therefore, by construction, $M' = M$. It is a call transition, so $Pr(\tau)(\mathsf{main})(pc) = \mathtt{call}(f)$, $\overline{q}'_{\mathsf{cl}}(\tau).\phi = \phi$, $\overline{q}'_{\mathsf{cl}}(\tau).\mathtt{pc_s} = \bot$ and $\overline{q}'_{\mathsf{cl}}(\tau).\mathtt{f} = \mathsf{main}$, and also $\overline{q}_{\mathsf{lib}}(\tau).\mathtt{pc} = 0$ and $\overline{q}_{\mathsf{lib}}(\tau).\phi = \phi$, $\overline{q}_{\mathsf{cl}}(\tau).\mathtt{pc_s} = pc + 1$ and $\overline{q}_{\mathsf{cl}}(\tau).\mathtt{f} = f$. By construction, $\overline{q}'(\tau) = \langle pc, \phi, \bot, \mathsf{main} \rangle$ and $\overline{q}(\tau) = \langle 0, \phi, pc + 1, f \rangle$. Then $\overline{q}'(\tau) \xrightarrow{\mathtt{CALL}(f,\phi)}_{Pr(\tau)} \overline{q}(\tau)$ and, therefore, $\overline{q}' \xrightarrow{\tau, \mathtt{CALL}(f,\phi)}_{Pr} \overline{q}$. Since the transition is program-internal, $\langle \overline{q}', M' \rangle \xrightarrow{\tau, \mathtt{CALL}(f,\phi)}_{Pr[L] \bowtie \mathsf{PSC}} \langle \overline{q}, M \rangle$.

By the induction hypothesis $\textsc{Compose}(t'_{\mathsf{cl}}, t'_{\mathsf{lib}})$, for $\langle \overline{q}'_{\mathsf{cl}}, M'_{\mathsf{cl}} \rangle$, $\langle \overline{q}'_{\mathsf{lib}}, M'_{\mathsf{lib}} \rangle$ there there exists t' such that $\mathsf{H}(t') = \mathsf{H}(t'_{\mathsf{cl}})$ and $\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t'}_{Pr[L] \bowtie \mathsf{PSC}} \langle \overline{q}', M' \rangle$. It is easy to see that $\mathsf{H}(t' \cdot \langle \tau, \mathtt{CALL}(f, \phi) \rangle) = \mathsf{H}(t'_{\mathsf{cl}} \cdot \langle \tau, \mathtt{CALL}(f, \phi) \rangle) = \mathsf{H}(t_{\mathsf{cl}})$, and we have shown that:

$$\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t'}_{Pr[L] \bowtie \mathsf{PSC}} \langle \overline{q}', M' \rangle \xrightarrow{\tau, \mathtt{CALL}(f,\phi)}_{Pr[L] \bowtie \mathsf{PSC}} \langle \overline{q}, M \rangle$$

To conclude the proof for this case, we let $t = t' \cdot \langle \tau, \mathtt{CALL}(f, \phi) \rangle$.

Suppose $\alpha = \alpha_{\mathsf{cl}} = \alpha_{\mathsf{lib}} = \langle \tau, \mathtt{RET}(f, \phi) \rangle$. Let $\overline{q}'_{\mathsf{lib}}(\tau).\mathtt{pc} = pc_{\mathsf{lib}}$ and $\overline{q}'_{\mathsf{cl}}(\tau).\mathtt{pc_s} = pc_{\mathsf{s}}$. The transition is program-internal, so $M'_{\mathsf{lib}} = M_{\mathsf{lib}}$ and $M'_{\mathsf{cl}} = M_{\mathsf{cl}}$; therefore, by construction, $M' = M$. It is a return transition, so $Pr(\tau)(f)(pc_{\mathsf{lib}}) = \mathtt{return}$, $\overline{q}'_{\mathsf{lib}}(\tau).\phi = \phi$ and $\overline{q}'_{\mathsf{cl}}(\tau).\mathtt{f} = f$, and also $\overline{q}_{\mathsf{cl}}(\tau).\mathtt{pc} = pc_{\mathsf{s}}$ and $\overline{q}_{\mathsf{cl}}(\tau).\phi = \phi$, $\overline{q}_{\mathsf{cl}}(\tau).\mathtt{pc_s} = \bot$ and $\overline{q}_{\mathsf{cl}}(\tau).\mathtt{f} = \mathsf{main}$. By construction, $\overline{q}'(\tau) = \langle pc_{\mathsf{lib}}, \phi, pc_{\mathsf{s}}, f \rangle$ and $\overline{q}(\tau) = \langle pc_{\mathsf{s}}, \phi, \bot, \mathsf{main} \rangle$. Then $\overline{q}'(\tau) \xrightarrow{\mathtt{RET}(f,\phi)}_{Pr(\tau)} \overline{q}(\tau)$, therefore, $\overline{q}' \xrightarrow{\tau, \mathtt{RET}(f,\phi)}_{Pr} \overline{q}$.

Since the transition is program-internal, $\langle \overline{q}', M' \rangle \xrightarrow{\tau, \mathtt{RET}(f,\phi)}_{Pr[L] \rtimes \mathsf{PSC}} \langle \overline{q}, M \rangle$.

By the induction hypothesis $\textsc{Compose}(t'_{\mathsf{cl}}, t'_{\mathsf{lib}})$, for $\langle \overline{q}'_{\mathsf{cl}}, M'_{\mathsf{cl}} \rangle$, $\langle \overline{q}'_{\mathsf{lib}}, M'_{\mathsf{lib}} \rangle$ there there exists t' such that $\mathsf{H}(t') = \mathsf{H}(t'_{\mathsf{cl}}) = \mathsf{H}_F(t'_{\mathsf{lib}})$ and $\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t'}_{Pr[L] \rtimes \mathsf{PSC}} \langle \overline{q}', M' \rangle$. It is easy to see that $\mathsf{H}(t' \cdot \langle \tau, \mathtt{RET}(f,\phi) \rangle) = \mathsf{H}(t'_{\mathsf{cl}} \cdot \langle \tau, \mathtt{RET}(f,\phi) \rangle) = \mathsf{H}(t_{\mathsf{cl}})$, and we have shown that:

$$\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t'}_{Pr[L] \rtimes \mathsf{PSC}} \langle \overline{q}', M' \rangle \xrightarrow{\tau, \mathtt{RET}(f,\phi)}_{Pr[L] \rtimes \mathsf{PSC}} \langle \overline{q}, M \rangle$$

To conclude the proof for this case, we let $t = t' \cdot \langle \tau, \mathtt{RET}(f,\phi) \rangle$.

Suppose $\alpha_{\mathsf{cl}} = \alpha_{\mathsf{lib}} = \alpha = \lightning$. From Eq. (III) we deduce that $\overline{q}_{\mathsf{cl}} = \overline{q}_{\mathsf{lib}} = \overline{q}_{\mathsf{Init}}$. By construction, $\overline{q} = \overline{q}_{\mathsf{Init}}$. We also deduce that $M'_{\mathsf{cl}} \xrightarrow{\lightning}_{\mathsf{PSC}} M_{\mathsf{cl}}$ and $M'_{\mathsf{lib}} \xrightarrow{\lightning}_{\mathsf{PSC}} M_{\mathsf{lib}}$ hold. We consider $M'_{\mathsf{cl}} \xrightarrow{\lightning}_{\mathsf{PSC}} M_{\mathsf{cl}}$ first. By Prop. 6, since L' is safe for Pr, $\mathsf{Var}(Pr \setminus F) \cap \mathsf{NVVar}$ separates M'_{cl}, which allows us to deduce $M'_{\mathsf{cl}}|_{\mathsf{Var}(Pr \setminus F)} \xrightarrow{\lightning}_{\mathsf{PSC}} M_{\mathsf{cl}}|_{\mathsf{Var}(Pr \setminus F)}$ by Lemma 1(4). From the induction hypothesis $\textsc{Compose}(t'_{\mathsf{cl}}, t'_{\mathsf{lib}})$, we know that $M' = \langle M'_{\mathsf{cl}}|_{\mathsf{Var}(Pr \setminus F)}, \mathsf{Var}(Pr \setminus F) \rangle \uplus \langle M'_{\mathsf{lib}}|_{\mathsf{Var}(L)}, \mathsf{Var}(L) \rangle$ Note that by Lemma 4(c), $M'|_{\mathsf{Var} \setminus \mathsf{Var}(L)} = M|_{\mathsf{Var} \setminus \mathsf{Var}(L)} = M_{\mathsf{cl}}|_{\mathsf{Var}(Pr \setminus F)}$. Therefore, we get $M'|_{\mathsf{Var} \setminus \mathsf{Var}(L)} \xrightarrow{\mathtt{per}}_{\mathsf{PSC}} M|_{\mathsf{Var} \setminus \mathsf{Var}(L)}$. We consider $M'_{\mathsf{lib}} \xrightarrow{\lightning}_{\mathsf{PSC}} M_{\mathsf{lib}}$ now. By Prop. 6, since L is safe for Pr', $\mathsf{Var}(L) \cap \mathsf{NVVar}$ separates M'_{lib}, which allows us to deduce $M'_{\mathsf{lib}}|_{\mathsf{Var}(L)} \xrightarrow{\lightning}_{\mathsf{PSC}} M_{\mathsf{lib}}|_{\mathsf{Var}(L)}$ by Lemma 1(4). From the induction hypothesis $\textsc{Compose}(t'_{\mathsf{cl}}, t'_{\mathsf{lib}})$, we know that $M' = \langle M'_{\mathsf{cl}}|_{\mathsf{Var}(Pr \setminus F)}, \mathsf{Var}(Pr \setminus F) \rangle \uplus \langle M'_{\mathsf{lib}}|_{\mathsf{Var}(L)}, \mathsf{Var}(L) \rangle$, so we obtain $M'|_{\mathsf{Var}(L)} \xrightarrow{\lightning}_{\mathsf{PSC}} M|_{\mathsf{Var}(L)}$. Overall, we have $M'|_{\mathsf{Var} \setminus \mathsf{Var}(Pr)} \xrightarrow{\lightning}_{\mathsf{PSC}} M|_{\mathsf{Var} \setminus \mathsf{Var}(Pr)}$ and $M'|_{\mathsf{Var}(L)} \xrightarrow{\lightning}_{\mathsf{PSC}} M|_{\mathsf{Var}(L)}$ hold. By Lemma 1(4), $M' \xrightarrow{\lightning}_{\mathsf{PSC}} M$, which gives us $\langle \overline{q}', M' \rangle \xrightarrow{\lightning}_{Pr[L] \rtimes \mathsf{PSC}} \langle \overline{q}, M \rangle$

By the induction hypothesis $\textsc{Compose}(t'_{\mathsf{cl}}, t'_{\mathsf{lib}})$, for $\langle \overline{q}'_{\mathsf{cl}}, M'_{\mathsf{cl}} \rangle$, $\langle \overline{q}'_{\mathsf{lib}}, M'_{\mathsf{lib}} \rangle$ there there exists t' such that $\mathsf{H}(t') = \mathsf{H}(t'_{\mathsf{cl}})$ and $\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t'}_{Pr[L] \rtimes \mathsf{PSC}} \langle \overline{q}', M' \rangle$. It is easy to see that $\mathsf{H}(t' \cdot \lightning) = \mathsf{H}(t'_{\mathsf{cl}} \cdot \lightning) = \mathsf{H}(t_{\mathsf{cl}})$, and we have shown that:

$$\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t'}_{Pr[L] \rtimes \mathsf{PSC}} \langle \overline{q}', M' \rangle \xrightarrow{\lightning}_{Pr[L] \rtimes \mathsf{PSC}} \langle \overline{q}, M \rangle$$

To conclude the proof for this case, we let $t = t' \cdot \lightning$.

Suppose $\alpha_{\mathsf{cl}} = \langle \tau, l_{\mathsf{cl}} \rangle$, $\alpha_{\mathsf{lib}} = \langle \tau, l_{\mathsf{lib}} \rangle$, and $l_{\mathsf{cl}}, l_{\mathsf{lib}} \in \mathsf{SFLab}$. Let us assume that $\overline{q}'(\tau).\mathtt{f} \in F$ (the case when $\overline{q}'(\tau).\mathtt{f} \notin F$ is analogous). We deduce from Eq. (III) that $\langle \overline{q}'_{\mathsf{lib}}(\tau).\mathsf{pc}, \overline{q}'_{\mathsf{lib}}(\tau).\phi \rangle \xrightarrow{l_{\mathsf{lib}}}_{L(\overline{q}'_{\mathsf{lib}}(\tau).\mathtt{f})} \langle \overline{q}_{\mathsf{lib}}(\tau).\mathsf{pc}, \overline{q}_{\mathsf{lib}}(\tau).\phi \rangle$ holds. By construction of $\overline{q}$ and $\overline{q}'$, and the concurrent program transition rules we obtain that $\overline{q}' \xrightarrow{\alpha_{\mathsf{lib}}}_{Pr} \overline{q}$ holds. We also deduce from Eq. (III) that $M'_{\mathsf{cl}} \xrightarrow{\alpha_{\mathsf{cl}}}_{\mathsf{PSC}} M_{\mathsf{cl}}$ and $M'_{\mathsf{lib}} \xrightarrow{\alpha_{\mathsf{lib}}}_{\mathsf{PSC}} M_{\mathsf{lib}}$. We first consider $M'_{\mathsf{cl}} \xrightarrow{\alpha_{\mathsf{cl}}}_{\mathsf{PSC}} M_{\mathsf{cl}}$. By Prop. 7, $\mathtt{varset}(\alpha_{\mathsf{cl}}) \subseteq \mathsf{Var}(L')$. Since L' is safe for Pr, firstly, $\mathtt{varset}(\alpha_{\mathsf{cl}}) \subseteq \mathsf{Var} \setminus \mathsf{Var}(Pr \setminus F)$, and secondly, by Propositions 6 and 8, $(\mathsf{Var} \setminus \mathsf{Var}(Pr \setminus F)) \cap \mathsf{NVVar}$ separates M'_{cl}, which allows us to deduce $M'_{\mathsf{cl}}|_{\mathsf{Var}(Pr \setminus F)} \xrightarrow{\tau, \mathtt{SF}}_{\mathsf{PSC}} M_{\mathsf{cl}}|_{\mathsf{Var}(Pr \setminus F)}$ by Lemma 1(2). From the induction hypothesis $\textsc{Compose}(t'_{\mathsf{cl}}, t'_{\mathsf{lib}})$, we know that $M' = \langle M'_{\mathsf{cl}}|_{\mathsf{Var}(Pr \setminus F)}, \mathsf{Var}(Pr \setminus F) \rangle \uplus \langle M'_{\mathsf{lib}}|_{\mathsf{Var}(L)}, \mathsf{Var}(L) \rangle$. Note that by Lemma 4(c), $M'|_{\mathsf{Var} \setminus \mathsf{Var}(L)} = M|_{\mathsf{Var} \setminus \mathsf{Var}(L)} = M_{\mathsf{cl}}|_{\mathsf{Var}(Pr \setminus F)}$. Therefore, we get $M'|_{\mathsf{Var} \setminus \mathsf{Var}(L)} \xrightarrow{\tau, \mathtt{SF}}_{\mathsf{PSC}} M|_{\mathsf{Var} \setminus \mathsf{Var}(L)}$. We now consider $M'_{\mathsf{lib}} \xrightarrow{\alpha_{\mathsf{lib}}}_{\mathsf{PSC}} M_{\mathsf{lib}}$. By Prop. 6, since L is safe for Pr', $\mathsf{Var}(L) \cap \mathsf{NVVar}$ separates M'_{lib}, which allows us to deduce $M'_{\mathsf{lib}}|_{\mathsf{Var}(L)} \xrightarrow{\alpha_{\mathsf{lib}}}_{\mathsf{PSC}} M_{\mathsf{lib}}|_{\mathsf{Var}(L)}$ by Lemma 1(2).

From the induction hypothesis $\textsc{Compose}(t'_{\mathsf{cl}}, t'_{\mathsf{lib}})$, we know that $M' = \langle M'_{\mathsf{cl}}|_{\mathsf{Var}(Pr \setminus F)}, \mathsf{Var}(Pr \setminus F)\rangle \uplus \langle M'_{\mathsf{lib}}|_{\mathsf{Var}(L)}, \mathsf{Var}(L)\rangle$, so we obtain $M'|_{\mathsf{Var}(L)} \xrightarrow{\alpha_{\mathsf{lib}}}_{\mathsf{PSC}} M|_{\mathsf{Var}(L)}$. Overall, $M'|_{\mathsf{Var}\setminus\mathsf{Var}(L)} \xrightarrow{\tau,\mathtt{SF}}_{\mathsf{PSC}} M|_{\mathsf{Var}\setminus\mathsf{Var}(L)}$ and $M'|_{\mathsf{Var}(L)} \xrightarrow{\alpha_{\mathsf{lib}}}_{\mathsf{PSC}} M|_{\mathsf{Var}(L)}$. Since $\langle \overline{q}', M'\rangle$ is reachable, by Prop. 6, we have that $\mathsf{Var}(L)$ separates M'. By Lemma 1 (2), $M' \xrightarrow{\alpha}_{\mathsf{PSC}} M$, which gives us $\langle \overline{q}', M'\rangle \xrightarrow{\alpha_{\mathsf{lib}}}_{Pr[L]\bowtie\mathsf{PSC}} \langle \overline{q}, M\rangle$.

By the induction hypothesis $\textsc{Compose}(t'_{\mathsf{cl}}, t'_{\mathsf{lib}})$, for $\langle \overline{q}'_{\mathsf{cl}}, M'_{\mathsf{cl}}\rangle$, $\langle \overline{q}'_{\mathsf{lib}}, M'_{\mathsf{lib}}\rangle$ there there exists t' such that $\mathsf{H}(t') = \mathsf{H}(t'_{\mathsf{cl}})$ and $\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}}\rangle \xrightarrow{t'}_{Pr[L]\bowtie\mathsf{PSC}} \langle \overline{q}', M'\rangle$. Recalling that $\mathsf{H}_F(\alpha_{\mathsf{cl}}) = \mathsf{H}_F(\alpha_{\mathsf{lib}})$, it is easy to see that $\mathsf{H}(t' \cdot \alpha_{\mathsf{lib}}) = \mathsf{H}(t'_{\mathsf{cl}} \cdot \alpha_{\mathsf{cl}}) = \mathsf{H}(t_{\mathsf{cl}})$, and we have shown that:

$$\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}}\rangle \xrightarrow{t'}_{Pr[L]\bowtie\mathsf{PSC}} \langle \overline{q}', M'\rangle \xrightarrow{\alpha_{\mathsf{lib}}}_{Pr[L]\bowtie\mathsf{PSC}} \langle \overline{q}, M\rangle$$

To conclude the proof for this case, we let $t = t' \cdot \alpha_{\mathsf{lib}}$.

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

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

Proof (Proof outline). The claim it proved by induction on n. For the induction step, we use the induction hypothesis with $MGC' = MGC[L_n^{\#}]$, and derive that $L_1 \uplus ... \uplus L_{n-1} \sqsubseteq_{MGC'} L_1^{\#} \uplus ... \uplus L_{n-1}^{\#}$. Then, we show that $MGC[L_1 \uplus ... \uplus L_{n-1}]$ correctly calls L_n w.r.t. $MGC[L_1^{\#} \uplus ... \uplus L_{n-1}^{\#}]$, and since have that $L_n \sqsubseteq_{MGC_n} L_n^{\#}$ for $MGC_n = MGC[L_1^{\#} \uplus ... \uplus L_{n-1}^{\#}]$, we can use the abstraction theorem to obtain that $\mathsf{H}(MGC[L_1 \uplus ... \uplus L_{n-1} \uplus L_n]) \subseteq \mathsf{H}(MGC[L_1 \uplus ... \uplus L_{n-1} \uplus L_n^{\#}])$. Together with the fact that $L_1 \uplus ... \uplus L_{n-1} \sqsubseteq_{MGC'} L_1^{\#} \uplus ... \uplus L_{n-1}^{\#}$, we obtain that $\mathsf{H}(MGC[L_1 \uplus ... \uplus L_{n-1} \uplus L_n]) \subseteq \mathsf{H}(MGC[L_1^{\#} \uplus ... \uplus L_{n-1}^{\#} \uplus L_n^{\#}])$, which concludes our proof.

Lemma 3. A trace t of $Pr \bowtie \mathsf{PSC}$ is called $\dot{m}_0$*-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 we have a binary relation R on $\mathsf{NVVar} \to \mathsf{Val}$ such that:

- $\langle \dot{m}_{\mathsf{Init}}, \dot{m}_{\mathsf{Init}}\rangle \in R$.
- 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 $L \sqsubseteq_{MGC} L^{\#}$.

Proof (Proof outline). Let $h \in \mathsf{H}(MGC[L])$. Let $h_1, ..., h_n$ be *crashless* histories such that $h = h_1 \cdot \lightning \cdot ... \cdot \lightning \cdot h_n$. Let $t_1, ..., t_n$ be crashless traces of $MGC[L] \bowtie \mathsf{PSC}$, such that $\mathsf{H}(t_i) = h_i$ for every $1 \leq i \leq n$. Let $\dot{m}_0, ..., \dot{m}_n$ be non-volatile memories such that each t_i is $\dot{m}_{i-1}$-to-$\dot{m}_i$. By repeatedly applying the assumption of the lemma (formally, inducting on n), we obtain a sequence of crashless traces $t_1^{\#}, ..., t_n^{\#}$ of $MGC[L^{\#}] \bowtie \mathsf{PSC}$ and non-volatile memories $\dot{m}_0^{\#}, ..., \dot{m}_n^{\#}$ such that each $t_i^{\#}$ is $\dot{m}_{i-1}^{\#}$-to-$\dot{m}_i^{\#}$ and satisfies $\mathsf{H}(t_i^{\#}) = h_i$. Then, it follows that $h = \mathsf{H}(t_1^{\#}) \cdot \lightning \cdot ... \cdot \lightning \cdot \mathsf{H}(t_n^{\#}) \in \mathsf{H}(MGC[L^{\#}])$.

Theorem 1 (Abstraction). Let libraries L and $L^{\#}$ and programs MGC and Pr be such that both L and $L^{\#}$ are safe for MGC and Pr, $L \sqsubseteq_{MGC} L^{\#}$ holds, and Pr correctly calls $L^{\#}$ w.r.t. MGC. Then, if $\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t}_{Pr[L] \bowtie \mathsf{PSC}} \langle \overline{q}, M \rangle$, 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 $\overline{q}^{\#}(\tau) = \overline{q}(\tau)$.
- $M^{\#}|_{\mathsf{Var}(Pr \setminus dom(L))} = M|_{\mathsf{Var}(Pr \setminus dom(L))}$.

Proof. Let $F = dom(L)$. It suffices to show $\mathsf{H}(Pr[L]) \subseteq \mathsf{H}(Pr[L^{\#}])$. Then, the claim follows using Lemma 2 (applied with $L := L^{\#}$, $L' := L$, $Pr := Pr$, 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 $\mathsf{H}(t) = h$. Let $\langle \overline{q}, M \rangle$ such that $\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t}_{Pr[L] \bowtie \mathsf{PSC}} \langle \overline{q}, M \rangle$. Clearly, t cannot be empty (since the empty history is a history of any program). Consider the last transition in t, and let t', α, and $\langle \overline{q}', M' \rangle$, such that $t = t' \cdot \alpha$ and $\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t'}_{Pr[L] \bowtie \mathsf{PSC}} \langle \overline{q}', M' \rangle \xrightarrow{\alpha}_{Pr[L] \bowtie \mathsf{PSC}} \langle \overline{q}, M \rangle$. Let $h' = \mathsf{H}(t')$. The minimality of t ensures that h' is a proper prefix of h, and thus α must correspond to a history label. In turn, using Prop. 1, the minimality of h ensures that $h' \in \mathsf{H}(Pr[L^{\#}])$.

Now, if $\alpha = \lightning$, then using Prop. 2, we would have $h = \mathsf{H}(t) = \mathsf{H}(t' \cdot \alpha) = h' \cdot \lightning \in \mathsf{H}(Pr[L^{\#}])$. Similarly, if $\alpha = \langle \tau, l \rangle$ for some $\tau \in \mathsf{Tid}$ and $l \in \mathsf{SFLab}$, then using Prop. 3, we would have $h = \mathsf{H}(t) = \mathsf{H}(t' \cdot \alpha) = h' \cdot \langle \tau, \mathtt{SF} \rangle \in \mathsf{H}(Pr[L^{\#}])$.

Hence, we have $\alpha = \langle \tau, \mathtt{CALL}(f, \phi) \rangle$ or $\alpha = \langle \tau, \mathtt{RET}(f, \phi) \rangle$ for some $\tau \in \mathsf{Tid}$, $f \in \mathsf{F}$, and $\phi : \mathsf{Reg} \to \mathsf{Val}$. It also follows that $\overline{q}' \xrightarrow{\alpha}_{Pr[L]} \overline{q}$ (since $\langle \overline{q}', M' \rangle \xrightarrow{\alpha}_{Pr[L] \bowtie \mathsf{PSC}} \langle \overline{q}, M \rangle$ and $\alpha \neq \mathtt{per}$).

We claim that $\overline{q}'(\tau).\mathtt{f} \in F$ (and so, it must be the case that $\alpha = \langle \tau, \mathtt{RET}(f, \phi) \rangle$ for $f \in F$). Indeed, suppose otherwise. Let $t'_{\#}$ and $\langle \overline{q}'_{\#}, M'_{\#} \rangle$ such that $\mathsf{H}(t'_{\#}) = h'$ and $\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t'_{\#}}_{Pr[L^{\#}] \bowtie \mathsf{PSC}} \langle \overline{q}'_{\#}, M'_{\#} \rangle$. Using Lemma 2 (applied with $L := L^{\#}$, $L' := L$, $Pr := Pr$, and $Pr' := Pr$), there exist $t''_{\#}$ and $\langle \overline{q}''_{\#}, M''_{\#} \rangle$ such that $\mathsf{H}(t''_{\#}) = h'$, $\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t''_{\#}}_{Pr[L^{\#}] \bowtie \mathsf{PSC}} \langle \overline{q}''_{\#}, M''_{\#} \rangle$, and $\overline{q}''_{\#}(\pi) = \overline{q}'(\pi)$ for every π such that $\overline{q}'(\pi).\mathtt{f} \notin F$. Now, since $\overline{q}' \xrightarrow{\alpha}_{Pr[L]} \overline{q}$ and $\overline{q}'(\tau).\mathtt{f} \notin F$, by Prop. 5, we have that $\overline{q}' \xrightarrow{\alpha}_{Pr[L^{\#}]} \overline{q}$. In addition, since $\overline{q}'(\tau).\mathtt{f} \notin F$, we also have $\overline{q}''_{\#}(\tau) = \overline{q}'_{\#}(\tau)$. Hence, α is enabled in $\overline{q}''_{\#}$ (in the LTS $Pr[L^{\#}]$), and so it is also enabled in $\langle \overline{q}''_{\#}, M''_{\#} \rangle$ (in the LTS $Pr[L^{\#}] \bowtie \mathsf{PSC}$). It follows that $t''_{\#} \cdot \alpha \in \mathsf{traces}(Pr[L^{\#}] \bowtie \mathsf{PSC})$, but since $\mathsf{H}(t''_{\#} \cdot \alpha) = h$, this contradicts the fact that $h \notin \mathsf{H}(Pr[L^{\#}])$.

Since Pr correctly calls $L^{\#}$ w.r.t. MGC, we have $\mathsf{H}_F(t'_{\#}) \in \mathsf{H}_F(MGC[L^{\#}])$. Let $t^*_{\#}$ and $\langle \overline{q}^*_{\#}, M^*_{\#} \rangle$ such that $\mathsf{H}_F(t^*_{\#}) = \mathsf{H}_F(t'_{\#})$ and $\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t^*_{\#}}_{MGC[L^{\#}] \bowtie \mathsf{PSC}} \langle \overline{q}^*_{\#}, M^*_{\#} \rangle$. Using Lemma 2 (applied with $L := L$, $L' := L^{\#}$, $Pr := MGC$, and $Pr' := Pr$), there exist t^* and $\langle \overline{q}^*, M^* \rangle$ such that $\mathsf{H}_F(t^*) = \mathsf{H}_F(t'_{\#})$, $\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t^*}_{MGC[L] \bowtie \mathsf{PSC}} \langle \overline{q}^*, M^* \rangle$, and $\overline{q}^*(\pi) = \langle \overline{q}'(\pi).\mathtt{pc}, \overline{q}'(\pi).\phi, \overline{q}^*_{\#}(\pi).\mathtt{pc_s}, \overline{q}^*_{\#}(\pi).\mathtt{f} \rangle$ for every π such that $\overline{q}^*_{\#}(\pi).\mathtt{f} \in F$. Since $\mathsf{H}_F(t^*) = \mathsf{H}_F(t'_{\#}) = \mathsf{H}_F(t')$ and $\overline{q}'(\tau).\mathtt{f} \in F$, by Prop. 4, we have $\overline{q}^*_{\#}(\tau).\mathtt{f} \in F$, and so $\overline{q}^*(\tau) = \langle \overline{q}'(\tau).\mathtt{pc}, \overline{q}'(\tau).\phi, \overline{q}^*_{\#}(\tau).\mathtt{pc_s}, \overline{q}^*_{\#}(\tau).\mathtt{f} \rangle$. Since $\overline{q}' \xrightarrow{\alpha}_{Pr[L]} \overline{q}$, it follows that α is enabled in $\overline{q}^*$ (in the LTS $MGC[L]$), and so it is also enabled in $\langle \overline{q}^*, M^* \rangle$ (in the LTS $MGC[L] \bowtie \mathsf{PSC}$).

Therefore, we have $t^* \cdot \alpha \in \mathsf{traces}(MGC[L] \bowtie \mathsf{PSC})$, and so $\mathsf{H}_F(t) = \mathsf{H}_F(t') \cdot \alpha = \mathsf{H}_F(t^*) \cdot \alpha = \mathsf{H}_F(t^* \cdot \alpha) \in \mathsf{H}_F(MGC[L])$. Then, the assumption that $L \sqsubseteq_{MGC} L^\#$, ensures that $\mathsf{H}_F(t) \in \mathsf{H}_F(MGC[L^\#])$.

Finally, using Lemma 2 (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.

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 $MGC_i = MGC[L_1^\# \uplus \ldots \uplus L_{i-1}^\# \uplus L_{i+1}^\# \uplus \ldots \uplus L_n^\#]$.

Proof. We prove the claim by induction on n. For $n = 1$, the claim trivially follows.

For the induction step, let $L_1, \ldots, L_n, L_1^\#, \ldots, L_n^\#$ be libraries, and let MGC be a program satisfying the required conditions. For $MGC' = MGC[L_n^\#]$, we have that

$$\mathsf{Var}(L_1), \ldots, \mathsf{Var}(L_{n-1}), \mathsf{Var}(L_1^\#), \ldots, \mathsf{Var}(L_{n-1}^\#), \mathsf{Var}(MGC' \setminus dom(L_1 \uplus \ldots \uplus L_{n-1}))$$

are pairwise disjoint. In addition, for every $1 \leq i \leq n-1$,

$$\begin{aligned} MGC'[L_1^\# \uplus \ldots \uplus L_{i-1}^\# \uplus L_{i+1}^\# \uplus \ldots \uplus L_{n-1}^\#] &= MGC[L_n^\#][L_1^\# \uplus \ldots \uplus L_{i-1}^\# \uplus L_{i+1}^\# \uplus \ldots \uplus L_{n-1}^\#] \\ &= MGC[L_1^\# \uplus \ldots \uplus L_{i-1}^\# \uplus L_{i+1}^\# \uplus \ldots \uplus L_n^\#] \end{aligned}$$

Hence, for every $1 \leq i \leq n-1$, we have $L_i \sqsubseteq_{MGC_i} L_i^\#$ for $MGC_i = MGC'[L_1^\# \uplus \ldots \uplus L_{i-1}^\# \uplus L_{i+1}^\# \uplus \ldots \uplus L_{n-1}^\#]$. By the induction hypothesis, it follows that $L_1 \uplus \ldots \uplus L_{n-1} \sqsubseteq_{MGC'} L_1^\# \uplus \ldots \uplus L_{n-1}^\#$.

Let $L = L_1 \uplus \ldots \uplus L_{n-1}$ and $L^\# = L_1^\# \uplus \ldots \uplus L_{n-1}^\#$. Then, we have $L \sqsubseteq_{MGC'} L^\#$, which implies that $\mathsf{H}(MGC[L \uplus L_n^\#]) \subseteq \mathsf{H}(MGC[L^\# \uplus L_n^\#])$. The latter implies that $MGC[L]$ correctly calls L_n w.r.t. $MGC[L^\#]$. In addition, by assumption we have $L_n \sqsubseteq_{MGC_n} L_n^\#$ for $MGC_n = MGC[L^\#]$. Hence, the abstraction theorem ensures that $\mathsf{H}(MGC[L \uplus L_n]) \subseteq \mathsf{H}(MGC[L \uplus L_n^\#])$. Together with the fact that $\mathsf{H}(MGC[L \uplus L_n^\#]) \subseteq \mathsf{H}(MGC[L^\# \uplus L_n^\#])$, we obtain that $\mathsf{H}(MGC[L \uplus L_n]) \subseteq \mathsf{H}(MGC[L^\# \uplus L_n^\#])$, which implies that $L \uplus L_n \sqsubseteq_{MGC} L^\# \uplus L_n^\#$, and concludes our proof.

Theorem 2. $L_{\mathrm{pair}} \sqsubseteq_{MGC_{\mathsf{rec}}} L_{\mathrm{pair}}^\#$.

Proof sketch. We use Lemma 3 to prove the claim. We let $\langle \dot{m}, \dot{m}^\# \rangle \in R$ iff the following hold:

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

Clearly, we have $\langle \dot{m}_{\mathsf{Init}}, \dot{m}_{\mathsf{Init}} \rangle \in R$. Suppose that $\langle \dot{m}_0, \dot{m}_0^\# \rangle \in R$. Let t be an $\dot{m}_0$-to-$\dot{m}$ crashless trace of $MGC_{\mathsf{rec}}[L_{\mathrm{pair}}] \bowtie \mathsf{PSC}$. We show that there exist a non-volatile memory $\dot{m}^\#$ and an $\dot{m}_0^\#$-to-$\dot{m}^\#$ crashless trace $t^\#$ of $MGC_{\mathsf{rec}}[L_{\mathrm{pair}}^\#] \bowtie \mathsf{PSC}$, such that $\langle \dot{m}, \dot{m}^\# \rangle \in R$ and $\mathsf{H}(t) = \mathsf{H}(t^\#)$. First, if t ends during the execution of the recovery method, then we obtain $t^\#$ by executing the call of the recovery

method, and take $\dot{m}^{\#} = \dot{m}_0^{\#}$. Otherwise, if recovery has completed, then after its completion, the invariant ensures that $M(\dot{\mathtt{x}}_1) = M^{\#}(\dot{\mathtt{x}}_1)$, $M(\dot{\mathtt{x}}_2) = M^{\#}(\dot{\mathtt{x}}_2)$, and $M(\dot{\mathtt{s}}) = 0$. Now, when the states are matching, by reusing the standard linearizability proof for the seqlock algorithm (see [8]), we can obtain a trace of $MGC_{\mathsf{rec}}[L^{\#}_{\mathrm{pair}}] \bowtie \mathsf{PSC}$ with the same history as t. It remains to handle persistency related steps, i.e., to decide when persist the block in the run of $L^{\#}$, in a way that establishes the required relation on the non-volatile memories in the end of the trace. For all complete executions of the write method, we persist the specification block just before the step in which the $\mathtt{fl}(\dot{\mathtt{x}}_1)$ is executed. For the incomplete invocations of the write method, we first note that at most one of them may manage to acquire the lock and persist an odd value of $\dot{\mathtt{s}}$ (the rest are waiting in the busy loop, and have nothing to persist). For that invocation, we persist the block at the point corresponding to the step in which the implementation persists the odd value of $\dot{\mathtt{s}}$. (Note that this mean that we may need to exclude the $\mathtt{fl}(\dot{\mathtt{x}}_1)$-step from the specification trace, and we can do so since the invocation did not complete.) This construction ensures that R holds for the non-volatile memories in the end of the trace. □

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

Proof (Proof sketch). We use Lemma 3 to prove the claim. We let $\langle \dot{m}, \dot{m}^{\#} \rangle \in R$ iff 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 $\dot{m}(\dot{\mathtt{x}}_2^{\mathtt{prev}}) = \dot{m}^{\#}(\dot{\mathtt{x}}_2)$.

Clearly, we have $\langle \dot{m}_{\mathsf{Init}}, \dot{m}_{\mathsf{Init}} \rangle \in R$. Suppose that $\langle \dot{m}_0, \dot{m}_0^{\#} \rangle \in R$. Let t be an $\dot{m}_0$-to-$\dot{m}$ crashless trace of $MGC_{\mathsf{rec}}[L_{\mathrm{bpair}}] \bowtie \mathsf{PSC}$. We show that there exist a non-volatile memory $\dot{m}^{\#}$ and an $\dot{m}_0^{\#}$-to-$\dot{m}^{\#}$ crashless trace $t^{\#}$ of $MGC_{\mathsf{rec}}[L^{\#}_{\mathrm{bpair}}] \bowtie \mathsf{PSC}$, such that $\langle \dot{m}, \dot{m}^{\#} \rangle \in R$ and $\mathsf{H}(t) = \mathsf{H}(t^{\#})$. First, if t ends during the execution of the recovery method, then we obtain $t^{\#}$ by executing the call of the recovery method, and take $\dot{m}^{\#} = \dot{m}_0^{\#}$. Otherwise, if recovery has completed, then after its completion, the invariant ensures that $M(\tilde{\mathtt{x}}_1) = M^{\#}(\dot{\mathtt{x}}_1)$ and $M(\tilde{\mathtt{x}}_2) = M^{\#}(\dot{\mathtt{x}}_2)$. In addition, since $\tilde{\mathtt{s}}$ is volatile, we also have $M(\tilde{\mathtt{s}}) = 0$. Now, when the states are matching, by reusing the standard linearizability proof for the seqlock algorithm (see [8]), we can obtain a trace of $MGC_{\mathsf{rec}}[L^{\#}_{\mathrm{bpair}}] \bowtie \mathsf{PSC}$ with the same history as t (in particular, note that the read an flush methods do not interfere whatsoever). It remains to handle persistency related steps, i.e., to decide when persist the block in the run of $L^{\#}$, in a way that establishes the required relation on the non-volatile memories in the end of the trace. Our construction performs all these persists just before the $\mathtt{fl}(\dot{\mathtt{x}}_1)$-step from the specification trace (when the flush method is executed). If there are incomplete invocations of the flush method in t, we first note that at most one of them may manage to acquire the lock and persist 0 for $\dot{\mathtt{f}}$ (the rest are waiting in the busy loop, and have nothing to persist). For that invocation, we persist the block at the point corresponding to the step in which the implementation persists 0 for $\dot{\mathtt{f}}$. (Note that this mean that we may need to exclude the $\mathtt{fl}(\dot{\mathtt{x}}_1)$-step from the specification trace, and we can do so since the invocation did not complete.) This construction ensures that R

holds for the non-volatile memories in the end of the trace. To show this, one shows that R is in fact an invariant of this construction that holds whenever the lock is not held ($M(\dot{\mathtt{f}}) = 0$).

D Additional remarks for Section 8

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 $L^{\#}_{S\not\downarrow \mathrm{b}}$ which is obtained from a sequential implementation S by wrapping each method of S inside a global lock and a persistence 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). 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 $L^{\#}_{S\not\downarrow \mathrm{b}}$. 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 $L^{\#}_{S_1 \cup S_2 \not\downarrow \mathrm{b}} = L^{\#}_{S_1 \not\downarrow \mathrm{b}} \cup L^{\#}_{S_2 \not\downarrow \mathrm{b}}$).