A Fully Verified Persistency Library*

Stefan Bodenmüller[1], John Derrick[2], Brijesh Dongol[3], Gerhard Schellhorn[1], and Heike Wehrheim[4]

[1] University of Augsburg, Augsburg, Germany
[2] University of Sheffield, Sheffield, UK
[3] University of Surrey, Guildford, UK
[4] University of Oldenburg, Oldenburg, Germany

Abstract. Non-volatile memory (NVM) technologies offer DRAM-like speeds with the added benefit of failure resilience. However, developing concurrent programs for NVM can be challenging since programmers must consider both inter-thread synchronisation and durability aspects at the same time. To alleviate this, libraries such as FliT have been developed to manage *transformations to durability*, allowing a linearizable concurrent object to be converted into a durably linearizable one by replacing the reads/writes to memory by calls to corresponding operations of the FliT library. However, a formal proof of correctness for FliT is missing, and standard proof techniques for durable linearizability are challenging to apply, since FliT itself is not durably linearizable. In this paper, we study the problem of proving correctness of transformations to durability. First, we develop an abstract *persistency library* (called PLib) that operationally characterises transformations to durability. We prove soundness of PLib via a forward simulation coupled with a *prophecy variable* used as an oracle about future behaviour. Second, we show correctness of the library FliT by proving that FliT*refines* PLib under the realistic PTSO memory model, i.e., the persistent version of TSO memory model implemented by Intel architectures. The proof of refinement between FliT and PLib has been mechanised within the theorem prover KIV. Taken together, these proofs guarantee that FliT is also sound wrt transformations to durability.

Keywords: Durable Linearizability, Px86-TSO, Persistency Libraries, FliT, Verification

1 Introduction

Byte-addressable *non-volatile (aka persistent) memory (NVM)*, e.g., Memory-Semantic SSD [25] and XL-FLASH [7], offer higher performance than flash memory while providing resilience to system-wide crashes. Unlike DRAM, the

* Dongol is supported by EPSRC grants EP/X037142/1, EP/X015149/1, EP/V038915/1, and EP/R025134/2. Wehrheim is supported by DFG project WE 2290/14-1.

contents of NVM survive failures (e.g., a power outage). However, developing NVM programs is challenging since they must provide properties such as failure atomicity and recoverability, introducing a high level of additional programmer overhead. This is particularly acute for concurrent programs under weak memory, which have thread-safety and concurrency synchronisation requirements in addition to persistency for failure atomicity.

In recent years, there have been several proposals [5, 13, 26, 30] that aim to help programmers with the transition to NVM, i.e., correct durability of concurrent programs developed for systems without NVM. Many of these approaches provide mechanisms that support *transformations to durability*, whereby program operations such as reads and writes to memory are replaced by calls to library operations that manage durability. In particular, given a (concurrent) library that is correct in a setting *without* crashes, the transformation mechanism can be used to ensure correctness *with* crashes. We note that the notion of "correctness" may also change as part of the transformation to durability.

Proposals such as TL4x and PMDK [1,26] support durable transactions that provide failure atomicity (but not necessarily thread safety [26]). Atlas [5] ensures thread-safe durability provided that a programmer can guarantee that the original program is race-free. Mirror [13] transforms linearizable [14] objects (viz., data structures like stacks or queues) into durably linearizable [15] objects, provided that the object in question is non-blocking. FliT [30] is a generic proposal that ensures correct durability by tracking read-write dependencies on behalf of a programmer using persistent-write-back (`pwb`) instructions. Neither FliT nor Mirror have a fully formalised correctness proof of such transformations.

In this paper, we take a more abstract approach to the question of transformations to durability for linearizable concurrent objects, which may implement concurrent data structures such as stacks or queues. As our first contribution, we develop a *persistency library* (called PLib) that abstractly models the essential requirements for durability. Such requirements capture the key *dependencies* between read and write accesses to shared variables and their persistence order. We prove that PLib guarantees sound transformation to durability. This proof proceeds by showing PLib refines a *canonical durable automaton* [9] that is derived from the sequential specification of the object. Our proof employs forward simulation using information about the future alike *prophecy variables* [8].

Our second contribution is a proof of correctness of the FliT library [30]. This proof leverages correctness of PLib by showing FliT in turn to refine PLib. This part of our overall soundness proof has been mechanised in the theorem prover KIV [28]. For this concrete implementation level, we model FliT over a real processor, i.e., to be subject to the effects of the (weak) memory model of the processor. Here, we take the memory model of persistent PTSO [16], the persistent version of the TSO memory model of Intel's x86 processor [23]. As a result, both refinements together show that FliT guarantees correct transformations to durability on PTSO.

Overview. This paper is structured as follows. §2 motivates our work and §3 presents the formal background and key definitions. We present PLib and its

```
put(x, arg){
  write(x.val, arg);
  write(x.flag, true);
}
```

```
get(x){
  r := read(x.flag)
  if r {
    s := read(x.val);
    return s; }
  return ⊥; }
```

Fig. 1: Operations `put` and `get` (initially `x.val` $= 0$ and `x.flag` $=$ *false*)

correctness proof in §4, present the FliT library implementation (under the PTSO memory model) in §5 and its correctness proof in §6.

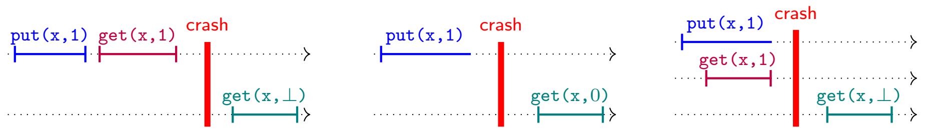
Auxiliary material. The KIV mechanisation corresponding to the proofs in §6 may be found at [4].

2 Motivation

We start by illustrating the challenges involved in concurrent programming on NVM. Consider the object O implementing `put` and `get` operations in Fig. 1 that use a variable `x.flag` to determine whether a value for a given location `x.val` has been set. Operation `get(x)` only reads from `x.val` after `x.flag` has been set, otherwise it returns the value ⊥[5]. Without NVM, these operations are clearly *linearizable* [14] wrt an abstract specification that performs each of the operations atomically since the fine-grained implementation only sets `x.flag` to true after writing to `x.val`. Thus any (concurrent) history of `put` and `get` operations can be reordered (while preserving the real-time order of operations) into a valid sequential history.

Let us now consider the behavior of Fig. 1 under NVM. In NVM, writes are first cached in *volatile memory*. Written values only reach *persistent memory* on flush actions. We then talk about *persisted writes*. Read actions return values from volatile memory and – if this does not contain an entry for a location – then read from persistent memory. On a crash, only the contents of persistent memory is preserved; thus a crash action resets the volatile memory to persistent memory. It is straightforward to see that executing `put` and `get` in such a memory model results in histories that are not *durably linearizable* [15], where durable linearizability holds if a history after removing crashes is linearizable. In particular, it is possible to generate the history in Fig. 2a, where the `get` operation before the crash can read from unpersisted writes of the `put` operation, and the crash can occur before these writes are persisted (so that the get after the crash returns ⊥). This history is not linearizable after removing the crash, thus is not durably linearizable. Similarly, since writes do not persist in

[5] Note that the `put` operation would in principle need to reset `flag` to *false* before writing a new value. For simplicity we elide this here.



(a) History H_1: Fails durably linearizability because the second `get` returns $\perp$ instead of `1`

(b) History H_2: Fails durably linearizability because the `get` returns 0 instead of `1` or $\perp$

(c) History H_3: Fails durably linearizability because the second `get` returns $\perp$ instead of `1`

Fig. 2: Three histories (dotted lines denote operations of different threads, x-axis is time)

the order of their occurrence, histories such as H_2 (Fig. 2b) would also be possible. Here, the `get` operation performs a read and sees the flag to be *true* (i.e., `x.flag` is persisted), but still returns the old (initial) value. A naive approach to addressing this issue is to modify the program so that each write is flushed to persistent memory immediately after it occurs. However, this approach is not only inefficient, it is also incorrect. For instance, under concurrency, it would be possible to generate the history in Fig. 2c since the second write to `x.flag` in `put` may have occurred, but not yet persisted, yet be read by the first `get`.

The key to achieving durable linearizability generically is to ensure that a client (i.e., a thread) executing an object's operation (e.g., `get`) that reads from another client operation (e.g., `put`) should *not* complete unless it can be assured that all writes it has read from have persisted. With this approach, histories such as H_3 (Fig. 2c) *cannot* occur even when the `put(x,1)` executes concurrently with `get(x)`. In particular, the `get` itself would then ensure that any unpersisted writes that it has read from have been persisted *before* returning. Thus it would be impossible for the second `get(x)` to return $\perp$. Second, to prevent writes from being persisted out-of-order, we must ensure prior writes executed by a thread have persisted before starting a new write by that thread. This prevents histories such as H_2 (Fig. 2b).

The ideas above have been implemented by libraries such as FliT [30] by providing read/write methods that efficiently track read-write dependencies between client operations. A programmer can gain durable linearizability guarantees by calling reads/writes of such a library in place of standard reads and writes, together with a call to a complete method before returning from an operation. The method complete ensures that any dependent unpersisted writes are persisted after executing complete. One of our main contributions in this paper is an abstract persistency library PLib (presented as an input/output automaton in Fig. 5) that we prove to abstractly capture this correctness principle. With this correctness proof for PLib at hand, we can then prove correctness of FliT by showing FliT to *refine* PLib.

$$\begin{tikzcd} as \arrow[r, "a"] \arrow[d, dash, dashed, "abs"] & \exists\ as' \arrow[d, dash, dashed, "abs"] \\ cs \arrow[r, "a"] & cs' \end{tikzcd} \qquad \begin{tikzcd} as \arrow[r, "\exists\ c"] \arrow[d, dash, dashed, "abs"] & \exists\ as' \arrow[d, dash, dashed, "abs"] \\ cs \arrow[r, "b"] & cs' \end{tikzcd} \qquad \begin{tikzcd} as \arrow[r, "\varepsilon"] \arrow[d, dash, dashed, "abs"] & as \arrow[d, dash, dashed, "abs"] \\ cs \arrow[r, "b"] & cs' \end{tikzcd}$$

Fig. 3: Possible cases for a forward simulation abs, where $a \in external(C) = external(A)$, $b \in internal(C)$, $c \in internal(A)$, as, as' are abstract and cs, cs' concrete states. ε indicates no action of A. Existentially quantified states and actions must be found such that the diagram commutes.

3 Using IOAs to Verify (Durable) Linearizability

This section provides some standard definitions and known results, providing background for the rest of the paper.

Input/Output Automata (IOA). We use input/output automata (IOA) [20] for all of our sequential and concurrent models. We let Loc be the set of shared locations, Tid the set of thread identifiers and Val the values of locations.

Definition 1 (Input/Output Automaton (IOA)). *An* Input/Output Automaton (IOA) *is a labeled transition system A with a set of* states $states(A)$*, a set of* actions $acts(A)$*, a set of* start states $start(A) \subseteq states(A)$*, and a* transition relation $trans(A) \subseteq states(A) \times acts(A) \times states(A)$ *(so that the actions label the transitions).*

The set $acts(A)$ is partitioned into internal actions $internal(A)$ that represent events of the system that are hidden from the environment, and external actions $external(A)$ (typically atomic steps or invocations/responses to calling an operation) representing the IOA's interactions with its environment. We specify IOAs (like the one in Fig. 4) by giving the states in terms of state variables and their initial values. For every action $a \in acts(A)$ we give a precondition Pre on the state s that enables a step $(s, a, s') \in trans(A)$, and specify the result state s' by assignments given under Eff.

We next define the correctness conditions relevant for our approach. Linearizability is the standard correctness condition for concurrent objects [14]. Here, we provide a definition of linearizability in terms of refinement.

Refinement. An *execution* of an IOA A is a sequence $\sigma = s_0 a_0 s_1 a_1 \dots s_n a_n s_{n+1}$ of alternating states and actions, such that $s_0 \in start(A)$ and for all states s_i, $(s_i, a_i, s_{i+1}) \in trans(A)$. We write $first(\sigma) = s_0$ for the initial and $last(\sigma)$ for the last state (if it exists) of an execution σ.

A *trace* of A (an element of $traces(A)$) is any sequence of (external) actions obtained by projecting the external actions of any execution of A. For IOAs C and A with $external(C) = external(A)$, we say that C is a *refinement* of A, denoted $C \leq A$, iff $traces(C) \subseteq traces(A)$. Refinement can be proven by establishing forward or backward simulations between IOAs (see e.g., [18]). Given that the methodology is well-known, we elide the formal definition, and provide an overview of the proof obligations for forward simulation diagrammatically in Fig. 3.

Canonical (Durable) Automata. We provide an operational definition of linearizability and durable linearizability using the notion of canonical (durable) automata, both of which are defined in terms of a sequential object specifying the expected behavior.

Definition 2 (Sequential Object). *A* sequential object *is a 4-tuple* (Σ, S, I, ρ) *where* Σ *is an* alphabet *of operations,* S *is a set of states and* $I \subseteq S$ *are the initial states, and* $\rho : (S \times \Sigma \times Val^*) \to 2^{S \times Val^*}$ *is a* transition relation *generating a set of next states and output values (if any) for a given state, operation and input values.*

The *canonical automaton* [19], $\textsc{Can}(\mathbb{S})$, and *canonical durable automaton* [9], $\textsc{DAut}(\mathbb{S})$, for a sequential object $\mathbb{S}$ are shown in Fig. 4. $\textsc{Can}(\mathbb{S})$ splits every operation $op \in \Sigma$ of $\mathbb{S}$ into separate external *invocation* and *response* actions and an internal *do*-action that occurs between the invocation and response. $\textsc{DAut}(\mathbb{S})$ additionally includes a crash action. The traces of a canonical (durable) automaton are well-formed *histories* [14, 15], comprising invocations and responses of each operation plus the crash actions in the case of the durable automaton. Unlike the sequential object, the histories of a canonical automaton may be concurrent. However, since the effect of operations *op* of the sequential object (the do-action) occurs between the invocation and response, each history of the canonical (durable) automaton is (durably) linearizable wrt the given sequential specification, $\mathbb{S}$. Thus, in general, an automaton is (durably) linearizable wrt $\mathbb{S}$ iff it refines the canonical (durable) automaton for $\mathbb{S}$. Note that the durable automaton reflects the *persistent* state of the object, thus, only the program counters of the threads are affected by the crash.

Definition 3 (Linearizability and Durable Linearizability). *Let* C *be an IOA. We say that* C *is* linearizable *wrt* $\mathbb{S}$ *if* $C \leq \textsc{Can}(\mathbb{S})$*, and that* C *is* durably linearizable *wrt* $\mathbb{S}$ *if* $C \leq \textsc{DAut}(\mathbb{S})$*.*

Our main result is to show that a *linearizable object* can be transformed into a *durably linearizable object.* Formally, this requires (c.f. [2]) that we transform an IOA whose histories contain invocations and responses of operations to histories that contain invocations, responses and crashes. The addition of crash is straightforward if the crash has no effect on the original IOA except on the program counters of the threads. Given an IOA A, we let A_{c} be the IOA A augmented with the crash action from Fig. 4. Note that $\textsc{DAut}(\mathbb{S}) = \textsc{Can}(\mathbb{S})_{\mathsf{c}}$. We have the following proposition.

Proposition 1. *If* $A \leq \textsc{Can}(\mathbb{S})$ *then* $A_{\mathsf{c}} \leq \textsc{DAut}(\mathbb{S})$*.*

Composing Client and Library IOAs. In the following, we will have several instances of a client calling a library, written *Client*[*Lib*]. These communicate via external actions, which may be *invocation* and *response* actions or atomic calls.

State variables:

$pc : \mathit{Tid} \to \{\mathit{notStarted}, \mathit{crashed}, \mathit{idle}\} \cup \{\mathsf{inv}(op), \mathsf{res}(op) \mid op \in \Sigma\}$	(initially: $\forall\, \tau \in \mathit{Tid}.\ pc(\tau) = \mathit{notStarted}$)
$s : S$	(initially: $s \in I$)
$in, out : \mathit{Tid} \to \mathit{Val}^*$	

Actions:

$\mathsf{inv}_\tau(op, v)$	$\mathsf{do}_\tau(op)$	$\mathsf{res}_\tau(op, v)$
Pre: $pc(\tau) \in \{\mathit{notStarted}, \mathit{idle}\}$	Pre: $pc(\tau) = \mathsf{inv}(op)$	Pre: $pc(\tau) = \mathsf{res}(op)$
	$(s', o') \in \rho(s, op, in(\tau))$	$v = out(\tau)$
Eff: $pc(\tau) := \mathsf{inv}(op)$	Eff: $pc(\tau) := \mathsf{res}(op)$	Eff: $pc(\tau) := \mathit{idle}$
$in(\tau) := v$	$(s, out(\tau)) := (s', o')$	

crash
Eff: $pc := \lambda\, \tau : \mathit{Tid}.\ \textbf{if}\ pc(\tau) \neq \mathit{notStarted}\ \textbf{then}\ \mathit{crashed}\ \textbf{else}\ pc(\tau)$

Fig. 4: The canonical durable automaton $\text{DAUT}(\mathbb{S})$ for a sequential object $\mathbb{S} = (\Sigma, S, I, \rho)$; automaton $\text{CAN}(\mathbb{S})$ is derived by removing the highlighted program counter values and transitions

The instances of *Client*[*Lib*] that we use are shown in Fig. 8. In [2], *Client*[*Lib*] is defined formally as a product of two IOAs and the following theorem is shown, which establishes refinement for client-library compositions.

Theorem 1 (Refinement in context). *Let A, B, C be IOAs with $\mathit{external}(C) = \mathit{external}(A) \subseteq \mathit{acts}(B)$. If $C \leq A$, then $B[C] \leq B[A]$.*

4 Abstract Persistency Library and its Correctness

In this section, we present our first contribution: the abstract persistency library PLib, which guarantees the transformation to durability for concurrent linearizable objects. An object O has to call the libWrite and libRead actions of PLib to access shared variables and additionally call complete before returning from operations. The key requirement on PLib to ensuring durability is that it tracks certain dependencies (see [30]) and persists writes in this order. Namely **1)** writes of the same thread must persist in the order in which they occur, and **2)** when a thread, say τ, executes complete, all of τ's writes as well as the writes τ has read from must have been persisted.

Persistency Library. Fig. 5 gives the IOA of PLib. The state of PLib assumes a volatile memory *vmem*, a persistent memory *pmem* and a *log* of all *read/write events* together with their persistency status and a program counter. Note that the program counter tracks the control flow of the *library* and is unrelated to the program counter of the client thread. When a crash occurs, all currently running threads are stopped and cannot continue. To comply with durable linearizability [15], a thread that crashes is never restarted. Note that like the canonical (durable) automata, PLib specifies a concurrent system since each thread can independently execute its actions.

$$
\begin{aligned}
pdr(x, \varepsilon) &= \varepsilon \\
pdr(x, \langle \mathsf{W}_\tau(x, v), \mathit{flag}\rangle \cdot log) &= \langle \mathsf{W}_\tau(x, v), \mathit{flag}\rangle \cdot log \\
pdr(x, \langle \mathsf{R}_\tau(x, v), \mathit{flag}\rangle \cdot log) &= \langle \mathsf{R}_\tau(x, v), \mathit{true}\rangle \cdot pdr(x, log) \\
pdr(x, \langle \mathsf{W}_\tau(x', v), \mathit{flag}\rangle \cdot log) &= \langle \mathsf{W}_\tau(x', v), \mathit{flag}\rangle \cdot pdr(x, log) \quad \text{for} \quad x \neq x' \\
pdr(x, \langle \mathsf{R}_\tau(x', v), \mathit{flag}\rangle \cdot log) &= \langle \mathsf{R}_\tau(x', v), \mathit{flag}\rangle \cdot pdr(x, log) \quad \text{for} \quad x \neq x'
\end{aligned}
$$

$$lwp(x, log) = \mathit{false} \leftrightarrow \exists\, log_1, log_2.\ log = log_1 \cdot \langle \mathsf{W}_\tau(x, v), \mathit{false}\rangle \cdot log_2 \wedge log_{2|\mathsf{Wr},x} = \varepsilon$$

State variables:

$vmem, pmem : Loc \to Val$	(initially: $\forall\, x \in Loc.\ vmem(x) = pmem(x) = 0$)
$log : ((\mathsf{Rd} \cup \mathsf{Wr}) \times \mathbb{B})^*$	(initially: $log = \varepsilon$)
$pc : Tid \to \{notStarted, idle, crashed\}$	(initially: $\forall\, \tau \in Tid.\ pc(\tau) = notStarted$)

Actions:

$\mathsf{libWrite}_\tau(x, v)$
Pre: $pc(\tau) \in \{notStarted, idle\}$
$log_{|\tau,\mathit{false}} = \varepsilon$
Eff: $vmem(x) := v$
$log := log \cdot \langle \mathsf{W}_\tau(x, v), \mathit{false}\rangle$
$pc(\tau) := idle$

$\mathsf{libRead}_\tau(x, v)$
Pre: $pc(\tau) \in \{notStarted, idle\}$
$v = vmem(x)$
Eff: $log := log \cdot \langle \mathsf{R}_\tau(x, v), lwp(x, log)\rangle$
$pc(\tau) := idle$

$\mathsf{complete}_\tau$
Pre: $pc(\tau) \in \{notStarted, idle\}$
$log_{|\tau,\mathit{false}} = \varepsilon$
Eff: $pc(\tau) := idle$

$\mathsf{persist}(x)$
Pre: $\exists\, log_1, log_2.$
$log = log_1 \cdot \langle \mathsf{W}_\tau(x, v), \mathit{false}\rangle \cdot log_2$
$\wedge\, log_{1|\mathsf{Wr},x,\mathit{false}} = \varepsilon$
Eff: $pmem(x) := v$
$log := log_1 \cdot \langle \mathsf{W}_\tau(x, v), \mathit{true}\rangle \cdot pdr(x, log_2)$

crash
Eff: $pc := \lambda\, \tau \in Tid.\ \textbf{if}\ pc(\tau) \neq notStarted\ \textbf{then}\ crashed\ \textbf{else}\ pc(\tau)$
$vmem := pmem$
$log := \varepsilon$

Fig. 5: PLib specification (where $\mathsf{log}_{|\mathsf{Wr}}$ restricts log to all Wr entries, $\mathsf{log}_{|\tau,\mathit{false}}$ to all entries of τ with flag *false* and so on)

Let Rd and Wr be the set of all read and write events. Each read event has the form $\mathsf{R}_\tau(x, v)$, where τ is a thread identifier, $x \in Loc$ is a location and $v \in Val$ is a value. Similarly, a write event has the form $\mathsf{W}_\tau(x, v)$. The *log* pairs write events with a boolean flag that determines whether the operation has been persisted. We call a write *persisted* if its effect is written to persistent memory and it has flag *true*, and a read *persisted* if the write that it reads from is persisted.

A libWrite and complete executed by thread τ may only proceed if it has neither unpersisted writes nor unpersisted reads in *log*, i.e. when $log_{|\tau,\mathit{false}} = \varepsilon$. We believe the condition is as weak as possible to allow a proof of durable

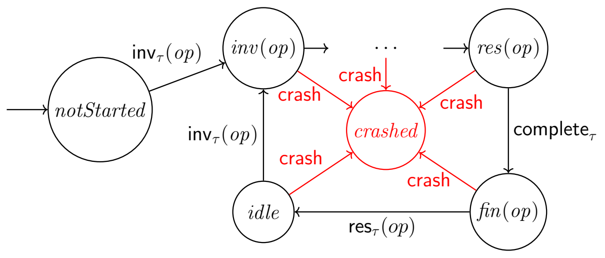


Fig. 6: States of a thread τ when calling operations of object O_{cc} ($\cdots$ are program counter values within the body of some $op \in \Sigma$)

$\mathsf{complete}_\tau$
Pre: $pc(\tau) = res(op)$
Eff: $pc(\tau) := fin(op)$

$\mathsf{res}_\tau(op)$
Pre: $pc(\tau) = fin(op)$
Eff: $pc(\tau) := idle$

Fig. 7: complete and res actions of O_{cc}

linearizability for arbitrary data structures: A write is disallowed only if there is a previous read or write by the same thread that has yet to be persisted. Returning from an operation must have persisted its values.

Execution of the write updates the value of the written location in *vmem*, then appends a new write event to *log* with flag *false*. Execution of read reads the value from volatile memory. It adds the read to the log together with a flag that is computed by checking whether the $\underline{l}$ast $\underline{w}$rite to this location (if there is any; otherwise the flag is set to true) has already been $\underline{p}$ersisted using predicate *lwp*. Persisting a write also $\underline{p}$ersists $\underline{d}$ependent $\underline{r}$eads (i.e., those which have read this value) using function *pdr*. The recursive definition of *pdr* ensures that the reads affected are those in the log before the next write to the same location.

***Correctness of* PLib.** Next, we prove theorem 2 below, i.e., that PLib guarantees a transformation to durability for linearizable objects. First, we explain the construction of the automaton $\mathsf{O}_{\mathsf{cc}}[\mathsf{PLib}]$. Let O_{cc} be the object O_{c} (as described in proposition 1) further modified so that each operation calls a complete action before returning (see Fig. 6). The complete and $\mathsf{res}(op)$ actions of O_{cc} are depicted in Fig. 7. Let $\mathsf{O}_{\mathsf{cc}}[\mathsf{PLib}]$ be O_{cc} but with internal actions read, write, complete and crash replaced with calls to the libRead, libWrite, complete and crash actions of PLib, as depicted on the left of Fig. 8.

Theorem 2. *Suppose* $\mathbb{S}$ *is a sequential specification and* O *an object. Then we have that* $\mathsf{O} \leq \textsc{Can}(\mathbb{S}) \Rightarrow \mathsf{O}_{\mathsf{cc}}[\mathsf{PLib}] \leq \textsc{DAut}(\mathbb{S})$.

We show that $\mathsf{O}_{\mathsf{cc}}[\mathsf{PLib}]$ is a refinement of $\textsc{DAut}(\mathbb{S})$ by proving three separate refinements:

1. $\mathsf{O}_{\mathsf{c}} \leq \textsc{DAut}(\mathbb{S})$. This follows from proposition 1 and the assumption $\mathsf{O} \leq \textsc{Can}(\mathbb{S})$.
2. $\mathsf{O}_{\mathsf{cc}} \leq \mathsf{O}_{\mathsf{c}}$. This change again only affects program counter values, thus it is trivial to show.
3. $\mathsf{O}_{\mathsf{cc}}[\mathsf{PLib}] \leq \mathsf{O}_{\mathsf{cc}}$. This is the challenging part, which we discuss in more detail below.

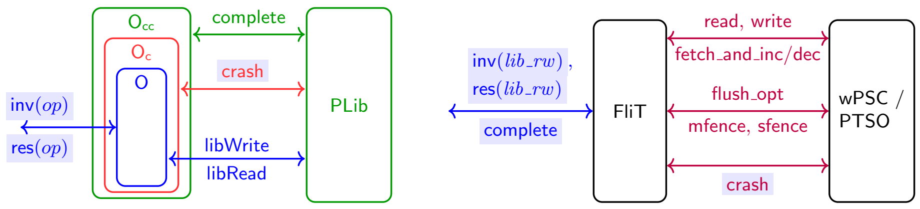


Fig. 8: Interface between client and library IOAs in our formal models; highlighted actions denote external events and $lib\_rw \in \{\mathsf{libRead}, \mathsf{libWrite}\}$

We start the proof of $\mathsf{O_{cc}}[\mathsf{PLib}] \le \mathsf{O_{cc}}$ by first clarifying the state and operations of $\mathsf{O_{cc}}$. As we need to show correctness of PLib for *all* objects, we cannot assume any specific knowledge about the actions of $\mathsf{O_{cc}}$. We only assume that $\mathsf{O_{cc}}$ uses the following state variables.

- A program counter opc for each thread τ, which in particular may take values $notStarted$, $idle$, $inv(op)$, $fin(op)$, $res(op)$ and $crashed$ as well as values for each atomic statement within the implementations of operations of the object op. The values are changed according to Fig. 6.
- The shared memory, $mem : Loc \to Val$, that is written atomically by write operations and read atomically by read operations.
- Local registers of type Reg for each thread and a mapping $regs : Reg \to Val$ that describes the local state that the object uses in its concurrent implementation of the sequential specification $\mathbb{S}$.

The state of $\mathsf{O_{cc}}[\mathsf{PLib}]$ is similar, except that $\mathsf{O_{cc}}[\mathsf{PLib}]$ calls libWrite/libRead of PLib instead of reading and writing to shared memory using the write and read operations. Thus, in $\mathsf{O_{cc}}[\mathsf{PLib}]$ the shared memory mem is omitted since memory is only indirectly accessed via PLib (see Fig. 5).

Next, we describe the overall idea of the proof. Proving refinement in our setting amounts to proving trace inclusion, i.e., $traces(\mathsf{O_{cc}}[\mathsf{PLib}]) \subseteq traces(\mathsf{O_{cc}})$. Concretely, from an execution of $\mathsf{O_{cc}}[\mathsf{PLib}]$ we will construct an execution of $\mathsf{O_{cc}}$ with the same sequence of external actions. This is guided by the following ideas:

Step 1. Crash events divide executions (and hence traces) of $\mathsf{O_{cc}}[\mathsf{PLib}]$ into so-called *eras* (of non-crash actions). We will reason about each era of $\mathsf{O_{cc}}[\mathsf{PLib}]$ in isolation and construct corresponding eras of $\mathsf{O_{cc}}$. At the end, we join the constructed eras of $\mathsf{O_{cc}}$ to form a consistent execution of $\mathsf{O_{cc}}$.

Step 2. Refinement is usually proven by showing a forward or backward simulation that uses an *abstraction relation* to relate states of a concrete automaton ($\mathsf{O_{cc}}[\mathsf{PLib}]$) to the states of an abstract automaton ($\mathsf{O_{cc}}$), see Fig. 3 for conditions in a forward simulation. Here, we will employ a forward simulation together with a *prophecy variable containing information about the future* to define the abstraction relation. Our prophecy variable refers to the log from the end of the era, which we use to determine whether libRead/libWrite actions at the concrete level should be matched to corresponding read/write actions at the abstract level of $\mathsf{O_{cc}}$, or to ε.

Step 3. For every execution σ of $\mathsf{O_{cc}}[\mathsf{PLib}]$ (more specifically, every era) we use forward simulation to show the existence of an execution ρ (viz. era) of $\mathsf{O_{cc}}$ such that $\rho_{|\tau} \preceq \sigma_{|\tau}$ holds for all threads τ. Basically, the construction of ρ keeps the steps of τ which are "persisted" in σ and preserves the external actions of the trace.

Step 1 (Decompose into eras). Let σ be an execution of $\mathsf{O_{cc}}[\mathsf{PLib}]$. Moreover, suppose $\sigma = \sigma_1\,\mathsf{crash}\,\sigma_2\,\mathsf{crash}\,\dots\,\mathsf{crash}\,\sigma_k$ such that each σ_i contains no crashes. We refer to $\sigma_1, \sigma_2, \dots, \sigma_k$ as the *eras* of σ.

Let σ_i be an era of a concrete execution of $\mathsf{O_{cc}}[\mathsf{PLib}]$. We construct a corresponding abstract era ρ_i of $\mathsf{O_{cc}}$ as follows. In σ_i, execution starts with $vmem = pmem$ since this is true initially and each crash resets $vmem$ to $pmem$ (see Fig. 5). The abstraction relation we use (see below) ensures that in corresponding states, mem of $\mathsf{O_{cc}}$ is the same as $pmem$. The crash actions for $\mathsf{O_{cc}}[\mathsf{PLib}]$ (and $\mathsf{O_{cc}}$) leaves $pmem$ (and mem) unchanged. The abstraction relation below further ensures that the same set of threads are "active" in corresponding concrete and abstract eras and hence the effect of crash on abstract and concrete program counters is the same. Thus $\rho_1\,\mathsf{crash}\,\rho_2\,\dots\,\mathsf{crash}\,\rho_n$ is an execution of $\mathsf{O_{cc}}$.

Step 2 (Constructing the abstraction relation). We augment the states of $\mathsf{O_{cc}}[\mathsf{PLib}]$ with information from the end of each era. More specifically, we consider the value $\mathsf{log} := s_n.log$ of the log variable at the end of the era $\sigma = s_0 a_0 s_1 a_1 \dots s_n$, and use it as a prophecy variable. (Note that although this discussion is about era σ_i, we drop the index i for notational convenience). Using the (standard) construction of [18, Proposition 5.13], log is added to every state s_j resulting in an era $\sigma' = s'_0 a_0 s'_1 a_1 \dots s'_n$.

We now work towards an abstraction relation that relates σ to a corresponding era $\rho = q_0 a'_0 q_1 a'_1 \dots q_n$ of $\mathsf{O_{cc}}$ by relating each state s'_j to q_j. We show that this relation gives rise to a forward simulation below. Note that, as described in Fig. 3, the forward simulation ensures that each concrete step is matched by some abstract step, thus σ and ρ have the same length. Namely, in the case of stuttering steps, the concrete action is matched by an abstract ε step.

By construction, in the abstract execution, $\mathsf{O_{cc}}$ can only perform reads and writes from the *persistent part* of log. Formally, the persistent part $pp(\mathsf{log})$ of log is defined recursively as a list of read and write events. We use $l(j) \neq \langle \mathsf{RW}_\tau(\_,\_), \mathit{false}\rangle$ to mean $l(j) \neq \langle \mathsf{W}_\tau(\_,\_), \mathit{false}\rangle \wedge l(j) \neq \langle \mathsf{R}_\tau(\_,\_), \mathit{false}\rangle$.

$$\begin{aligned} pp(\varepsilon) &= \varepsilon \\ pp(l \cdot \langle e_\tau, \mathit{false}\rangle) &= pp(l) \\ pp(l \cdot \langle e_\tau, \mathit{true}\rangle) &= \textbf{if } \forall\, j.\ l(j) \neq \langle \mathsf{RW}_\tau(\_,\_), \mathit{false}\rangle \textbf{ then } pp(l)\cdot e_\tau \textbf{ else } pp(l) \end{aligned}$$

Thus $pp(\mathsf{log})$ contains all writes in era σ that are persisted by the end of σ. Note that the precondition $\mathsf{libWrite}$ for PLib (Fig. 5) ensures that there cannot be an earlier write with flag false followed by a write from the same thread with flag true. Persisted reads are dropped from the persistent part iff they are preceded by other unpersisted reads or writes of the same thread.

The reads and writes executed in era σ' when we reach state s'_i are recorded in $s'_i.log$ (see Fig. 5). Operations that are persisted before the end of the era σ are those in the persistent part of $\mathsf{plog}(i) := \mathsf{log}[0..\#(s'_i.log) - 1]$, i.e., in $pp(\mathsf{plog}(i))$. Thus, $\mathsf{plog}(i)$ contains the same events as $s'_i.log$ in the same order. However, while $s'_i.log$ indicates read/writes that have been persisted thus far, $\mathsf{plog}(i)$ records the reads/writes that are guaranteed to be persisted before the era σ ends.

We can use $\mathsf{plog}(i)$ to relate the persistent memory $s'_i.pmem$ to the abstract memory $q_i.mem$. In q_i we want to have exactly the memory state that corresponds to executing the events of $pp(\mathsf{plog}(i))$, the relation between persistent memory of s'_i and the abstract memory $q_i.mem$ is thus

$$q_i.mem = s'_0.pmem \oplus pp(\mathsf{plog}(i)) \tag{1}$$

where $pmem \oplus evs$ is inductively defined as

$$\begin{aligned} pmem \oplus \varepsilon &= pmem \\ pmem \oplus ((\mathsf{R}_\_(\_,\_),\_) \cdot evs) &= pmem \oplus evs \\ pmem \oplus ((\mathsf{W}_\_(x,v),\_) \cdot evs) &= pmem[x \mapsto v] \oplus evs \end{aligned}$$

So the abstract memory $q_i.mem$ is obtained by taking the persistent memory $pmem$ from s'_0 and performing the updates of all writes of the persisted part of the log that have been executed up to state s'_i in order. Each individual thread τ has executed reads and writes that are persisted at the end of the era only, if and only if $\mathsf{plog}(i)_{|\tau}$ and $s_i.log_{|\tau}$ are equal, which is equivalent to $\mathsf{plog}(i)_{|\tau,false} = \varepsilon$.

The abstraction relation keeps registers and program counter values of a thread τ identical in the concrete and abstract levels until the thread reaches its first unpersisted read or write, i.e., until $\mathsf{plog}(i)_{|\tau,false}$ becomes nonempty. Therefore, the abstraction relation states

$$\mathsf{plog}(i)_{|\tau,false} = \varepsilon \implies s'_i.regs = q_i.regs \wedge s'_i.pc = q_i.pc \tag{2}$$

The full abstraction relation is the conjunction of (1) and (2).

Step 3 (Proving forward simulation). For the forward simulation, we have to map the steps $s'_i a_i s'_{i+1}$ of the augmented era σ' of $\mathsf{O_{cc}}[\mathsf{PLib}]$ to steps of the corresponding era in $\mathsf{O_{cc}}$ (in the sense of Fig. 3). Specifically,

– if $a_i = \mathsf{libWrite}(x,v)$ and $\mathsf{plog}(i+1)_{|\tau,false} = \varepsilon$ we map to $q_i\, \mathsf{write}(x,v)\, q_{i+1}$ in the abstract era ρ,
– if $a_i = \mathsf{libRead}(x,v)$ and $\mathsf{plog}(i+1)_{|\tau,false} = \varepsilon$ we map to $q_i\, \mathsf{read}(x,v)\, q_{i+1}$ in the abstract era ρ, and
– in all other cases, we map to $q_i\, \varepsilon\, q_{i+1}$ in ρ

Note that $\mathsf{plog}(i)_{|\tau,false} = \varepsilon$ iff $pp(\mathsf{plog}(i))_{|\tau} = \mathsf{plog}(i)_{|\tau}$.

Intuitively, a thread continues executing actions until it reaches a read or write that is not persisted at the end of the era (and hence will be lost when a crash occurs). The libReads and libWrites that are persisted in the concrete era are

```
libWrite(x, arg){
  sfence;
  fetch_and_inc(flit_ctr(x));
  write(x, arg);
  flush_opt(x);
  sfence;
  fetch_and_dec(flit_ctr(x));
}
```

```
libRead(x){
  val = read(x);
  if (flit_ctr(x) > 0)
    flush_opt(x);
  return val;
}

complete(){ mfence; }
```

Fig. 9: Core algorithm of the FliT library

mapped to reads and writes. Both sets of corresponding operations see the same memory state at the abstract and concrete levels due to (1) of the abstraction relation. Steps of the object that are preserved can be mapped one-to-one since they see the same register values by (2).

This mapping of steps preserves invocations and responses (i.e., gives us the same sequence of external actions): A thread executing a return (the argument for an invocation is similar) must have all reads and writes persisted in $s_i.log_{|\tau}$, since it has just called the complete operation. Therefore, all events are persisted in $\mathsf{plog}(i)_{|\tau}$ as well, so the step is kept in the abstract era. All earlier actions of the thread are kept as well, simply because $\mathsf{plog}(j)$ for $j \leq i$ is a prefix of $\mathsf{plog}(i)$.

5 The Library FliT and Memory Models

For our second contribution, we prove soundness (of a transformation to durability) for a concrete persistency library (FliT [30]) executing on a realistic memory model (PTSO [16, 24]). We first explain FliT and PTSO, then prove soundness in §6.

***Library* FliT.** The pseudo code of the core algorithm of FliT is given in Fig. 9. For our proofs, it is translated into an IOA as described in [10] with a step for every instruction. FliT comprises operations `libRead` and `libWrite` that perform reads from and writes to shared memory (i.e., locations `x` in *Loc*)[6]. Like $\mathsf{O_{cc}}[\mathsf{PLib}]$, the concurrent object $\mathsf{O_{cc}}$ calls `libRead` and `libWrite` when accessing memory. In addition, FliT implements an operation `complete` that the object O has to call before completing one of its operations $op \in \Sigma$.

The key idea behind FliT is that it manages persistent memory operations on behalf of a user. It guarantees that crucial dependencies among reads and writes are maintained and thereby achieves durable linearizability. The shared variable `flit_ctr(x)` for location `x` is used (for efficiency) to determine whether

[6] In [30], the operations are called `shared_load` and `shared_store`, respectively. Since we aim to prove a durable linearizability theorem that assumes all reads and writes are durable, we omit the additional `pflag` of the original algorithm and assume that each FliT operation is called with the persistent flag. We also do not make use of private stores and loads (this is future work).

a shared write is in progress, i.e., another thread is currently writing to location `x` so that the current value at `x` may not be persisted yet. `libWrite` performs a `fetch_and_inc` (i.e., a *read-modify-write*, RMW, operation) on `flit_ctr(x)` before performing the write itself, and a corresponding `fetch_and_dec` (another RMW) on `flit_ctr(x)` after the write. Both `libRead` and `libWrite` issue `flush_opt` instructions[7]. Finally, `libWrite` issues an `sfence` instruction at its start and end, and `complete` issues an `mfence`.[8] Basically, the `flush_opt` operation serves as a marker signifying a prior access of a thread τ to location `x`. RMW and `mfence`/`sfence` operations must wait until the marked access is persisted, i.e., reaches persistent memory.

Memory models. FliT achieves its guarantees by employing a number of hardware instructions with a semantics defined by the *memory model* of the processor. Here, we study the usage of FliT on the memory model TSO of the Intel x86 processor [21], viz. its version PTSO for NVM as formalized in [16]. There are alternative definitions of the persistent version of x86 (e.g., [6, 23]); we have taken PTSO here because it matches the intention of developers about program behaviour well and it has a clear connection to the persistent version of sequential consistency (which we will employ in our proof). Fig. 10 provides an automaton specification of PTSO based on the operational semantics given in [16], and Fig. 8 (on the right) shows the events FliT uses for communication with the memory model. For simplicity, we do not give a specification of `flush` instructions here, as they are not used by FliT. For the same reason, we omit the generic RMW transitions and directly define transitions for `fetch_and_inc` and `fetch_and_dec`.

PTSO contains non-volatile memory $m : Loc \to Val$ and shared persistence buffers $P(x)$ storing a sequence of markers $\mathsf{W}_\tau(v)$ and $\mathsf{FO}(\tau)$ for every $x \in Loc$, representing pending `write` and `flush_opt` instructions by thread τ that have not yet been persisted. These markers in P are visible to all threads, i.e., threads can access the values of pending writes in P. Note that, compared to the original model in [16], $\mathsf{W}_\tau(v)$ markers are augmented with the identifier τ of the thread that issued the write, which is relevant for verification only.

In addition to P, PTSO contains a private store buffer $B(\tau)$ for each thread τ that contains similar markers $\mathsf{W}(x, v)$ and $\mathsf{FO}(x)$ for `write` and `flush_opt` instructions, as well as markers SF issued by `sfence` instructions. When a `write`, `flush_opt`, or `sfence` is executed by a thread τ, the corresponding markers are first put into its store buffer $B(\tau)$ (actions write_τ, $\mathsf{flush\_opt}_\tau$, and sfence_τ in Fig. 10). *Propagate* steps then move entries from store buffers to persistence buffers (under some conditions), which make the entries visible to other threads (actions prop_w, prop_fo, and prop_sf). Finally, *persist* steps move entries from persistence buffers to the main memory m (actions persist_w and persist_fo). As

[7] In [30], they employ an instruction `pwb` (persistent-write-back) for this. Here, we directly give the architecture-specific instruction.

[8] In [30], both operations use a generic `PFENCE` instruction for which no precise semantics is given. We instantiate these instructions with the appropriate concrete fence instructions based on the semantics of the memory models given by [16, 24].

State variables:

$m : \mathit{Loc} \to \mathit{Val}$ (initially: $m = \lambda\ x.\ 0$)

$P : \mathit{Loc} \to \{\mathsf{FO}(\tau), \mathsf{W}_\tau(v) \mid \tau \in \mathit{Tid}, v \in \mathit{Val}\}^*$ (initially: $P = \lambda\ x.\ \varepsilon$)

$B : \mathit{Tid} \to \{\mathsf{FO}(x), \mathsf{W}(x, v), \mathsf{SF} \mid \tau \in \mathit{Tid}, x \in \mathit{Loc}, v \in \mathit{Val}\}^*$ (initially: $B = \lambda\ \tau.\ \varepsilon$)

Actions:

mfence_τ
Pre: $B(\tau) = \varepsilon$
$\forall\ y.\ \mathsf{FO}(\tau) \notin P(y)$

sfence_τ
Eff: $B(\tau) := B(\tau) \cdot \mathsf{SF}$

$\mathsf{persist\_fo}(x)$
Pre: $P(x) = \mathsf{FO}(\tau) \cdot p$
Eff: $P(x) := p$

$\mathsf{prop\_fo}$
Pre: $B(\tau) = b_1 \cdot \mathsf{FO}(x) \cdot b_2$
$\mathsf{W}(x, \_), \mathsf{FO}(x), \mathsf{SF} \notin b_1$
Eff: $B(\tau) := b_1 \cdot b_2$
$P(x) := P(x) \cdot \mathsf{FO}(\tau)$

$\mathsf{flush\_opt}_\tau(x)$
Eff: $B(\tau) := B(\tau) \cdot \mathsf{FO}(x)$

$\mathsf{persist\_w}(x)$
Pre: $P(x) = \mathsf{W}_\tau(v) \cdot p$
Eff: $P(x) := p$
$m(x) := v$

$\mathsf{prop\_w}$
Pre: $B(\tau) = \mathsf{W}(x, v) \cdot b$
Eff: $B(\tau) := b$
$P(x) := P(x) \cdot \mathsf{W}_\tau(v)$

$\mathsf{prop\_sf}$
Pre: $B(\tau) = \mathsf{SF} \cdot b$
$\forall\ y.\ \mathsf{FO}(\tau) \notin P(y)$
Eff: $B(\tau) := b$

$\mathsf{read}_\tau(x, v)$
Pre: $v = (m \oplus P \oplus B(\tau))(x)$

$\mathsf{write}_\tau(x, v)$
Eff: $B(\tau) := B(\tau) \cdot \mathsf{W}(x, v)$

$\mathsf{fetch\_and\_inc/dec}_\tau(\mathsf{x}, \mathsf{v})$
Pre: $B(\tau) = \varepsilon$
$\forall\ y.\ \mathsf{FO}(\tau) \notin P(y)$
$v = (m \oplus P \oplus B(\tau))(x)$
Eff: $P(x) := P(x) \cdot \mathsf{W}_\tau(v \pm 1)$

crash
Eff: $P := \lambda\ x.\ \varepsilon$
$B := \lambda\ \tau.\ \varepsilon$

Fig. 10: Automaton of the PTSO memory model (which is equivalent to Px86 [16]).

given by the crash action, all pending markers in B and P are lost during a crash (the buffers are reset to ε), while only m is kept.

When thread τ intends to read location x (action read_τ), it first of all consults its store buffer, then the persistence buffer and finally persistent memory. This is modelled by the following function producing the combined memory $m \oplus P \oplus B(\tau)$ visible to thread τ.

$$m \oplus P \oplus B(\tau) = \lambda\ x. \begin{cases} v & \text{if } B(\tau) = b_1 \cdot \mathsf{W}(x, v) \cdot b_2 \wedge \mathsf{W}(x, \_) \notin b_2 \\ v & \text{else if } \mathsf{W}(x, \_) \notin B(\tau) \wedge P(x) = p_1 \cdot \mathsf{W}_\_(v) \cdot p_2 \\ & \qquad\qquad \wedge\, \mathsf{W}_\_(v) \notin p_2 \\ m(x) & \text{otherwise} \end{cases}$$

The read-modify-write actions $\mathsf{fetch\_and\_inc}_\tau(\mathsf{x})$ and $\mathsf{fetch\_and\_dec}_\tau(\mathsf{x})$ block until the store buffer is empty, and immediately propagate the updated value to the shared persistence buffer. Additionally, they require that all FO-markers of τ have left the persistence buffers, so that they have a similar effect to the strong mfence_τ action.

We prove soundness of FliT via a chain of refinements. This chain includes an intermediate memory model called wPSC, as an abstraction of PTSO. This memory model is based on the often assumed *sequential consistency* model (SC [17]), resp. PSC for NVM as given in [16]. The automaton of wPSC is given in Fig. 11. wPSC abstracts from the store buffers B of PTSO so that all markers are directly put into the shared persistence buffers P. Consequently, wPSC does not have any *propagate* actions, and write_τ and $\mathsf{flush\_opt}_\tau$ steps directly add markers to $P(x)$. Actions sfence_τ and mfence_τ are identical in wPSC as they both require all FO-markers of τ to be persisted. The combined memory for reading is identical for all threads: $m \oplus P = m \oplus P \oplus \varepsilon$.

State variables:

$m : \mathit{Loc} \to \mathit{Val}$ (initially: $m = \lambda\ x.\ 0$)

$P : \mathit{Loc} \to \{\mathsf{FO}(\tau), \mathsf{W}_\tau(v) \mid \tau \in \mathit{Tid}, v \in \mathit{Val}\}^*$ (initially: $P = \lambda\ x.\ \varepsilon$)

Actions:

$\mathsf{mfence}_\tau/\mathsf{sfence}_\tau$
Pre: $\forall\ y.\ \mathsf{FO}(\tau) \notin P(y)$

weak persist_fo(x)
Pre: $P(x) = p_1 \cdot \mathsf{FO}(\tau) \cdot p_2$
$p_{1|\tau} = \varepsilon$ $p_{1|\mathsf{W}} = \varepsilon$
Eff: $P(x) := p_1 \cdot p_2$

crash
Eff: $P := \lambda\ x.\ \varepsilon$

$\mathsf{flush\_opt}_\tau(\mathsf{x})$
Eff: $P(x) := P(x) \cdot \mathsf{FO}(\tau)$

weak persist_w(x)
Pre: $P(x) = p_1 \cdot \mathsf{W}_\tau(v) \cdot p_2$
$p_{1|\tau} = \varepsilon$
$p_{1|\mathsf{W}} = \varepsilon$
Eff: $P(x) := p_1 \cdot p_2$
$m(x) := v$

$\mathsf{read}_\tau(x, v)$
Pre: $v = (m \oplus P)(x)$

$\mathsf{write}_\tau(x, v)$
Eff: $P(x) := P(x) \cdot \mathsf{W}_\tau(v)$

$\mathsf{fetch\_and\_inc/dec}_\tau(x, v)$
Pre: $\forall\ y.\ \mathsf{FO}(\tau) \notin P(y)$
$v = (m \oplus P)(x)$
Eff: $P(x) := P(x) \cdot \mathsf{W}_\tau(v \pm 1)$

Fig. 11: Memory models wPSC and PSC, where PSC is obtained by removing blue parts.

The motivation for using this intermediate layer is to split the proof into two parts dealing with separate concerns. Based on wPSC, the first part of the proof (see §6.1) shows that the use of `flush_opt` and `sfence` resp. `mfence` in FliT is sufficient to guarantee a persist order that coincides with the persist constraints of our specification PLib. The second part of the proof (see §6.2) then shows that, when introducing per-thread store buffers in PTSO, propagations may only lead to uncritical reorderings in the persistence buffers. However, this is only the case in the context of FliT (or a similar persistency library).

Fig. 11 also depicts the differences to the original PSC model of [16]. Basically, wPSC differs only in that the *persist* actions have slightly weaker preconditions than PSC: persisting an FO- or W-marker of location x is possible even if there are some other markers before it in $P(x)$ (as long as they are just FOs of other threads). This adjustment is necessary due to the possible propagation reorderings of PTSO, see also §6.2.

6 Proving Correctness of FliT

Our correctness proof of FliT proceeds via proving FliT to *refine* PLib, i.e., to satisfy the requirements that PLib imposes on persistence orderings. More specifically, we aim to prove $\mathsf{FliT}[\mathsf{PTSO}] \leq \mathsf{PLib}^{\mathsf{IR}}$. Therein, $\mathsf{PLib}^{\mathsf{IR}}$ (where IR = Invoke/Respond) is a slight modification of PLib which replaces libRead and libWrite with invocations to and responses from the library (i.e., inv(libRead), inv(libWrite), res(libRead), res(libWrite)) in addition to do-steps (i.e., do(libRead), do(libWrite)). This split is necessary because FliT's reads and writes do not occur atomically. The proof of $\mathsf{FliT}[\mathsf{PTSO}] \leq \mathsf{PLib}^{\mathsf{IR}}$ proceeds in two steps:

1. $\mathsf{FliT}[\mathsf{wPSC}] \leq \mathsf{PLib}^{\mathsf{IR}}$, where wPSC is given in Fig. 11.
2. $\mathsf{FliT}[\mathsf{PTSO}] \leq \mathsf{FliT}[\mathsf{wPSC}]$, where PTSO is given in Fig. 10.

Combining this result with the simple to prove refinement $\mathsf{PLib}^{\mathsf{IR}} \leq \mathsf{PLib}$ and theorems 1 and 2, we get correct transformation to durability for FliT on PTSO, i.e., $\mathsf{O} \leq \textsc{Can}(\mathbb{S}) \Rightarrow \mathsf{O}_{\mathsf{cc}}[\mathsf{FliT}[\mathsf{PTSO}]] \leq \textsc{DAut}(\mathbb{S})$.

The proofs of the two refinements are non-trivial; each of them took around three weeks to verify. They have been mechanised in the theorem prover KIV [28] and are described below. The proofs follow the strategy detailed in [10, 27], which splits invariants and the abstraction relation into a global part for the shared state and a thread-local part to get thread-local proof obligations.

6.1 Proof of $\mathsf{FliT}[\mathsf{wPSC}] \leq \mathsf{PLib}^{\mathsf{IR}}$

We show the two refinements via forward simulations. This means that we need to define an abstraction relation, *abs*, and then have to map steps (occurrences of actions) of the concrete automaton to steps of the abstract one as depicted in Fig. 3.

The first refinement $\mathsf{FliT}[\mathsf{wPSC}] \leq \mathsf{PLib}^{\mathsf{IR}}$ maps all relevant steps of $\mathsf{FliT}[\mathsf{wPSC}]$ (in particular, the inv, do, and res steps of read and write as well as the $\mathsf{persist\_w}$ step) steps to the corresponding steps of $\mathsf{PLib}^{\mathsf{IR}}$. The only exception is $\mathsf{persist\_fo}$ which is mapped to an empty step. The core of the verification then is to ensure that the precondition of $\mathsf{do}(\mathsf{write})$ of $\mathsf{FliT}[\mathsf{wPSC}]$ is more restrictive than the one of $\mathsf{PLib}^{\mathsf{IR}}$.

Essentially, the modifications of the flit counter (which are RMW instructions of wPSC) are responsible for this: the `fetch_and_inc` instruction before writing ensures that the values read before the write get persisted, the `fetch_and_dec` at the end of writing ensures that the write just done gets persisted, so each thread never has more than one unpersisted write (the formal proof has corresponding assertions). However, formally the preconditions of RMW instructions ($\mathsf{fetch\_and\_inc/dec}_\tau$ in Fig. 11) only guarantee that all FO-markers of the thread have left the persistence buffer $P(x)$, so some additional reasoning is needed to prove this. The crucial assertion for $\mathsf{FliT}[\mathsf{wPSC}]$ is that a thread τ that has just added an FO-marker to $P(x)$ before the `fetch_and_dec` instruction will keep this marker longer in $P(x)$ than the write event W it has added to $P(x)$ in the write instruction right before it. Also, the FO-marker will always be after the write event, so the write must be persisted first. To ensure this, all global (persist) steps and all steps by other threads than τ must preserve this. The thread relies on the fact that all these steps can only remove an initial piece from each $P(x)$ (including some writes) and then some of the FO-markers before the next write (but not FO-markers after that write). They can also add new events to the end of $P(x)$, but those are not events of thread τ.

The abstraction relation between $\mathsf{FliT}[\mathsf{wPSC}]$ and $\mathsf{PLib}^{\mathsf{IR}}$ keeps both the volatile and persistent memories equal (where the volatile memory of $\mathsf{FliT}[\mathsf{wPSC}]$ is $m \oplus P$). Not yet persisted events for any location x (the elements of $log_{|x,false}$) must have a corresponding entry in $P(x)$. For writes, the same write events must appear in the same order in $P(x)$. For reads, an FO-marker must appear later in $P(x)$ than their position in *log*, except if the thread executing the read is directly after the read instruction, before it has a chance to add an FO-marker.

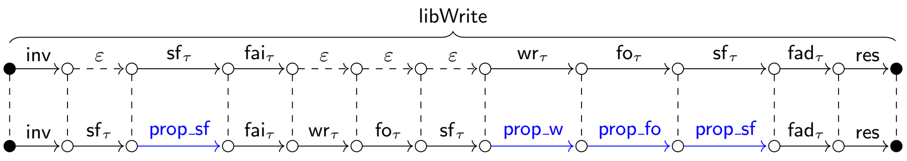


Fig. 12: Forward simulation (cf. Fig. 3) for FliT[PTSO] $\leq$ FliT[wPSC] for the libWrite operation. Internal steps of PTSO are depicted blue. wr$_\tau$, fo$_\tau$, sf$_\tau$, fad$_\tau$, fai$_\tau$ stand for write$_\tau$, flush_opt$_\tau$, sfence$_\tau$, fetch_and_dec$_\tau$, fetch_and_inc$_\tau$, respectively.

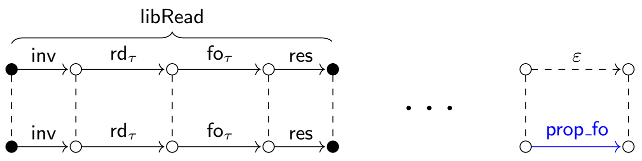


Fig. 13: Forward simulation (cf. Fig. 3) for FliT[PTSO] $\leq$ FliT[wPSC] for the libRead operation. Internal steps of PTSO are depicted blue. rd$_\tau$, fo$_\tau$ stand for read$_\tau$, flush_opt$_\tau$, respectively.

6.2 Proof of FliT[PTSO] $\leq$ FliT[wPSC]

The main idea for the proof of FliT[PTSO] $\leq$ FliT[wPSC] is to keep the two volatile memories represented by the two persistence buffers P (denoted P_{wPSC} and P_{PTSO} in the following) approximately equal. In particular, both should contain the same sequence of write events, ensuring that reading on both levels yields the same result values. The refinement therefore has to delay `write`s of FliT[PTSO], which add the event to the store buffer B, until they are propagated to P_{PTSO} by the system via a prop_w step. To get a forward simulation, the `flush_opt` and `sfence` instructions called during libWrite (see Fig. 9) are delayed, too. They take effect only when they are propagated to P_{PTSO}. Formally, in the forward simulation, they all refine an empty step ε of FliT[wPSC], while the propagation steps refine the instructions of FliT[wPSC] (see Fig. 12). The delay is possible since propagation is ensured to happen in the same order, and since the fetch_and_dec$_\tau$ step at the end of writing can proceed only when the events have all been propagated and the store buffer B is empty (so the responses from writing at the end can be matched again). Informally, the "linearization point" of libWrite of FliT[PTSO] is not the write instruction which adds to B, but the propagation of the write to volatile memory, and this is the order in which writes happen in FliT[wPSC] as well as in PLib$^{\mathsf{IR}}$.

For reading the situation is different. There, we cannot match adding an FO-marker (with flush_opt$_\tau$) on FliT[wPSC] to the corresponding prop_fo step on FliT[PTSO] since propagation may happen long after the read operation has finished. Therefore, it is necessary to match the flush_opt$_\tau$ steps one-to-one, and instead to match prop_fo for FO-markers issued by reading to an empty step ε

as shown in Fig. 13. The two instances of prop_fo for an FO(τ) can be distinguished by checking whether thread τ is currently in its libWrite. However, the abstraction relation must now deal with the reordering of FO-markers caused by this mapping, which results in the somewhat tricky abstraction relation that essentially allows FO-markers of reads that are in P_{wPSC} to be somewhere later in P_{PTSO} or still in B.

The persist steps (persist_fo and persist_w, which are omitted in Fig. 12) of FliT[wPSC] and FliT[PTSO] are mapped one-to-one since W-events must have the same order in P_{PTSO} as in P_{wPSC}. However, the weakening of the original PSC model to wPSC as shown in Fig. 11 is essential for this refinement: PTSO always persists the leading marker in P_{PTSO} for some location x, and thus, wPSC must be able to persist the corresponding event in P_{wPSC}. But due to the potential reordering of FO-markers (see above), this event does not have to be the leading event in P_{wPSC}. The weakened preconditions of the wPSC persist steps take this into account while not being too liberal (in particular, the order of writes must be maintained).

7 Related Work

D'Osualdo, Raad and Vafeiadis [11] pursue the idea of reusing proofs of linearizability for showing concrete data structures to be durable linearizable. They formulate a so-called *Pathway Theorem* which presents a method for proving durable linearizability. This method is modular in the sense of separating the proof of linearizability from issues concerning volatile and persistent memory. They base their work on a declaratively defined memory model. A similar approach (but for durable opacity) has been investigated by Bila et al. [2] who aim at reusing proofs of opacity [3]. Durable opacity is the NVM analogue of the correctness criterion *opacity* employed for Transactional Memory [29] (not concurrent objects). Bila et al. assume sequential consistency.

Verification of FliT differs from these earlier works [2, 10, 11] since FliT is a generic instrumentation technique. It transforms *any* linearizable object into a durably linearizable object. Thus, the proof methods in [2, 10, 11] are orthogonal to ours since they provide proof techniques for concurrent objects that are already durably linearizable *without* using FliT. Moreover, FliT alone is not durably linearizable.

Besides FliT, there are a number of other works investigating techniques for making linearizable concurrent objects durably linearizable, most notably NVTraverse [12] and Mirror [13]. Both provide transformations to durability for lock-free concurrent objects whereby NVTraverse is limited to node-based tree data structures. Mirror is trimmed towards using a volatile replica of the data structure in RAM for speeding up reading (though it can be used solely with NVM as well). Its approach for ensuring durable linearizability is stricter than the one of FliT: the algorithm enforces modifications to be persisted before their linearization points. To achieve this, reading accesses only the volatile replica, which is only updated after the corresponding write has passed the persistence buffer. Moreover, writing is performed via a strong CAS implementation, which prevents multiple pending writes to the same location in the persistence buffers. This behaviour also matches the PLib specification presented in this paper (yielding a *log* that always contains only persisted R markers and at most one unpersisted W marker per location). Thus, a formal proof of Mirror's correctness should be possible by showing a refinement analogous to the one for FliT shown in §6, which we intend to do as future work.

Israelevitz and Scott [15] also propose a technique for converting linearizable objects into (buffered) durably linearizable objects by inserting `pfence` and `pwb` instructions. They prove correctness of their transformations. Montage [31] is a library for use by programmers when intending to achieve buffered durable linearizability [15]. To this end, programmers need to identify so-called payloads and follow several additional rules. Neither of these approaches is accompanied by a fully formal proof of its soundness.

Regarding other memory models, recent works [6] have defined persistency models for Arm (PArm) building on a prior operational semantics based on promises [22]. To prove correctness of FliT under this model, one would need to introduce other types of `fence` and `flush` instructions (that are compatible with PArm), then prove that this implementation refines PLib. However, NVM implementations and semantics for Arm are less developed than for TSO; the PArm semantics by Cho et al. [6] has not been validated against real hardware. Moreover, the current implementation of FliT is for Px86 only[9]. We therefore see a full verification of FliT[PArm] against PLib to be future work. Such work would be able to reuse the correctness of PLib described in §4. The focus instead would be on developing appropriate abstractions, invariants, and refinement relations as described in §5 for FliT[PTSO].

8 Conclusion

In this paper, we have studied approaches for transformations to durability of linearizable objects. We have proposed an abstract persistency library and have proven that it guarantees such a transformation. As a second step, we have shown that one of the existing concrete libraries (FliT) also correctly implements such a transformation, in particular when running on the weak memory model PTSO. As future work, we intend to consider a full proof for other such proposals, specifically for Mirror [13].

Acknowledgements. We thank Michal Friedman for helping clarify the behaviour of FliT.

Data availability. The KIV specifications and mechanised proofs supporting this work's results are publicly available at figshare with the identifier https://doi.org/10.6084/m9.figshare.24132495.v1.

[9] See https://github.com/cmuparlay/flit.

References

1. G. Assa, A. Correia, P. Ramalhete, V. Schiavoni, and P. Felber. Tl4x: Buffered durable transactions on disk as fast as in memory. In M. M. Dehnavi, M. Kulkarni, and S. Krishnamoorthy, editors, *PPoPP*, pages 245–259. ACM, 2023. doi:10.1145/3572848.3577495.
2. E. Bila, J. Derrick, S. Doherty, B. Dongol, G. Schellhorn, and H. Wehrheim. Modularising verification of durable opacity. *Log. Methods Comput. Sci.*, 18(3), 2022. doi:10.46298/lmcs-18(3:7)2022.
3. E. Bila, S. Doherty, B. Dongol, J. Derrick, G. Schellhorn, and H. Wehrheim. Defining and Verifying Durable Opacity: Correctness for Persistent Software Transactional Memory. In A. Gotsman and A. Sokolova, editors, *FORTE*, volume 12136 of *LNCS*, pages 39–58. Springer, 2020. doi:10.1007/978-3-030-50086-3_3.
4. S. Bodenmüller, J. Derrick, B. Dongol, G. Schellhorn, and H. Wehrheim. Verification of FLIT Refinements – KIV proofs. https://kiv.isse.de/projects/FLIT.html. Executable artifact available at https://doi.org/10.6084/m9.figshare.24132495.v1, 2023.
5. D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. *ACM SIGPLAN Notices*, 49(10):433–452, 2014.
6. 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*, pages 16–31, 2021.
7. J. Choe. Review and things to know: Flash memory summit 2022. *TechInsights*, August 2022. URL: https://www.techinsights.com/blog/review-and-things-know-flash-memory-summit-2022.
8. W. P. de Roever and K. Engelhardt. *Data Refinement: Model-oriented Proof Theories and their Comparison*, volume 46 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.
9. J. Derrick, S. Doherty, B. Dongol, G. Schellhorn, and H. Wehrheim. Verifying correctness of persistent concurrent data structures. In M. H. ter Beek, A. McIver, and J. N. Oliveira, editors, *FM*, volume 11800 of *LNCS*, pages 179–195. Springer, 2019. doi:10.1007/978-3-030-30942-8_12.
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*, 33(4-5):547–573, 2021. URL: https://link.springer.com/article/10.1007/s00165-021-00541-8.
11. E. D'Osualdo, A. Raad, and V. Vafeiadis. The path to durable linearizability. *Proc. ACM Program. Lang.*, 7(POPL):748–774, 2023. doi:10.1145/3571219.
12. M. Friedman, N. Ben-David, Y. Wei, G. E. Blelloch, and E. Petrank. NVTraverse: in NVRAM data structures, the destination is more important than the journey. In A. F. Donaldson and E. Torlak, editors, *PLDI*, pages 377–392. ACM, 2020. doi:10.1145/3385412.3386031.
13. M. Friedman, E. Petrank, and P. Ramalhete. Mirror: making lock-free data structures persistent. In S. N. Freund and E. Yahav, editors, *PLDI*, pages 1218–1232. ACM, 2021. doi:10.1145/3453483.3454105.
14. M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, 1990.
15. J. Izraelevitz, H. Mendes, and M. L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In C. Gavoille and D. Ilcinkas, editors, *DISC*, volume 9888 of *Lecture Notes in Computer Science*, pages 313–327. Springer, 2016. doi:10.1007/978-3-662-53426-7_23.

16. A. Khyzha and O. Lahav. Taming x86-TSO persistency. *Proc. ACM Program. Lang.*, 5(POPL):1–29, 2021. doi:10.1145/3434328.
17. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.
18. N. Lynch and F. Vaandrager. Forward and backward simulations. *Information and Computation*, 121(2):214 – 233, 1995.
19. N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.
20. N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *PODC*, pages 137–151, New York, NY, USA, 1987. ACM.
21. S. Owens, S. Sarkar, and P. Sewell. A Better x86 Memory Model: x86-TSO. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 391–407. Springer, 2009. doi:10.1007/978-3-642-03359-9_27.
22. C. Pulte, J. Pichon-Pharabod, J. Kang, S.-H. Lee, and C.-K. Hur. Promising-ARM/RISC-V: a simpler and faster operational concurrency model. In K. S. McKinley and K. Fisher, editors, *PLDI*, pages 1–15. ACM, 2019. doi:10.1145/3314221.3314624.
23. A. Raad, J. Wickerson, G. Neiger, and V. Vafeiadis. Persistency semantics of the Intel-x86 architecture. *Proc. ACM Program. Lang.*, 4(POPL):11:1–11:31, 2020. doi:10.1145/3371079.
24. A. Raad, J. Wickerson, and V. Vafeiadis. Weak persistency semantics from the ground up: Formalising the persistency semantics of ARMv8 and transactional models. *PACMPL*, 3(OOPSLA):135:1–135:27, 2019.
25. Samsung Electronics. Samsung electronics unveils far-reaching, next-generation memory solutions at flash memory summit 2022, August 2022. https://news.samsung.com/global/samsung-electronics-unveils-far-reaching-next-generation-memory-solutions-at-flash-memory-summit-2022.
26. S. Scargall. *Programming persistent memory: A comprehensive guide for developers.* Springer Nature, 2020.
27. G. Schellhorn, S. Bodenmüller, and W. Reif. Thread-Local, Step-Local Proof Obligations for Refinement of State-Based Concurrent Systems. In *ABZ 2023: Rigorous State-Based Methods*, volume 14010 of *LNCS*, 2023.
28. G. Schellhorn, S. Bodenmüller, M. Bitterlich, and W. Reif. Software & System Verification with KIV. In *The Logic of Software. A Tasting Menu of Formal Methods*, volume 13360 of *LNCS*, page 408–436. Springer, 2022.
29. N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.
30. Y. Wei, N. Ben-David, M. Friedman, G. E. Blelloch, and E. Petrank. FliT: a library for simple and efficient persistent algorithms. In J. Lee, K. Agrawal, and M. F. Spear, editors, *PPoPP*, pages 309–321. ACM, 2022. doi:10.1145/3503221.3508436.
31. H. Wen, W. Cai, M. Du, L. Jenkins, B. Valpey, and M. L. Scott. A fast, general system for buffered persistent data structures. In X.-H. Sun, S. Shende, L. V. Kalé, and Y. Chen, editors, *ICPP*, pages 73:1–73:11. ACM, 2021. doi:10.1145/3472456.3472458.