Formal Methods in System Design
https://doi.org/10.1007/s10703-024-00462-1

RESEARCH

A verified durable transactional mutex lock for persistent x86-TSO

Eleni Vafeiadi Bila[1,2] · Brijesh Dongol[1]

Received: 5 November 2023 / Accepted: 19 July 2024

Abstract
The advent of non-volatile memory technologies has spurred intensive research interest in correctness and programmability. This paper addresses both by developing and verifying a durable (aka persistent) transactional memory (TM) algorithm, $\text{dTML}_{\text{Px86}}$. Correctness of $\text{dTML}_{\text{Px86}}$ is judged in terms of *durable opacity*, which ensures both *failure atomicity* (ensuring memory consistency after a crash) and *opacity* (ensuring thread safety). We assume a realistic execution model, Px86, which represents Intel's persistent memory model and extends the *Total Store Order* memory model with instructions that control persistency. Our TM algorithm, $\text{dTML}_{\text{Px86}}$, is an adaptation of an existing software transactional mutex lock, but with additional synchronisation mechanisms to cope with Px86. Our correctness proof is operational and comprises two distinct types of proofs: (1) proofs of invariants of $\text{dTML}_{\text{Px86}}$ and (2) a proof of refinement against an operational specification that guarantees durable opacity. To achieve (1), we build on recent Owicki–Gries logics for Px86, and for (2) we use a simulation-based proof technique, which, as far as we are aware, is the first application of simulation-based proofs for Px86 programs. Our entire development has been mechanised in the Isabelle/HOL proof assistant.

Keywords Persistent memory · Transactional memory · Verification · Refinement · Isabelle/HOL

1 Introduction

Non-volatile memory (NVM) technologies, e.g., Intel Optane, enable byte-addressable accesses as allowed by DRAM, while retaining the benefits of persistent storage. NVM has the potential to radically impact future systems since they can be designed to efficiently *recover* from a system-wide crash. However, NVM also introduces new programming challenges and requires previous notions of correctness to be reconsidered. Such challenges are particularly acute for concurrent programs, where one additionally has to understand inter-

✉ Brijesh Dongol
b.dongol@surrey.ac.uk

Eleni Vafeiadi Bila
eleni.vafeiadibila@arm.com

1 University of Surrey, Guildford, UK

2 Arm Ltd, Cambridge, UK

actions between *persistency* (concerning the order in which memory updates are persisted) and *weak memory consistency* (concerning the order in which memory updates in one thread become visible to other threads).

There has been widespread interest on NVM, with several works characterising their semantics in the context of *hardware* weak memory models [12, 57, 59, 60]. Alongside these low-level semantics, a separate line of work has focussed on adapting correctness conditions such as *linearisability* [34] and *opacity* [33], obtaining corresponding conditions such as *durable linearisability* [38] and *durable opacity* [5]. Such conditions provide a basis for developing high-level synchronisation mechanisms such as concurrent objects, in the case of (durable) linearisability, and transactional memory, in the case of (durable) opacity.

Our work brings these two lines of work together in the context of *recoverable concurrent transactional memory* [14, 39, 41, 47]. In particular, we develop dTML$_{\text{Px86}}$, which is an adaptation of the *durable* Transactional Mutex Lock (dTML) algorithm [5], which is itself a durable extension of the Transactional Mutex Lock [15] with logging mechanisms that support recoverability. The dTML algorithm has been designed for a strong memory model (PSC) [44], which extends sequential consistency (SC) [48] with persistency. However, PSC is a strong memory model that is unrealistic for modern architectures (e.g., Intel), which only provide weak memory guarantees.

1.1 Designing, modelling and verifying *dTML$_{Px86}$*

Unlike prior works, as the name implies, dTML$_{\text{Px86}}$ assumes Intel's x86 persistency and consistency model (Px86) [37], which extends the x86 Total Store Order (TSO) model [64] with a persistency semantics [12, 44, 57, 59, 60]. Here, like in TSO, each write is first cached in a local *FIFO store buffer* (and only visible to the writing thread), then later propagated to the volatile shared memory (whereby it becomes visible to other threads). Writes in the volatile shared memory are later *persisted* by propagating them to NVM.

In Px86, the order in which writes become persistent may differ from the order in which they were issued. To address this, Px86 provides instructions, e.g., **flush**, **flush**$_{\text{opt}}$, that explicitly flush locations[1] to NVM, ensuring that the corresponding locations are persisted. The **flush** instruction flushes a location line in a synchronous manner, blocking the executing thread until the prior write has been persisted. The *optimised flush* instruction (**flush**$_{\text{opt}}$) flushes a single location but in an asynchronous manner (without blocking the execution of the corresponding thread). The **flush**$_{\text{opt}}$ instruction is not ordered with respect to any following write, **flush**$_{\text{opt}}$, or **flush** (when applied to an address in a different location) instructions [57, Fig. 3], and only serves to *tag* locations that are to be persisted later. As a result, the execution of a **flush**$_{\text{opt}}$ on an address x, does not provide any guarantees about the value of x in persistent memory. To restrict the additional weak behaviors that **flush**$_{\text{opt}}$ introduces, Px86 provides a *store fence* (**sfence**) instruction that orders store instructions with **flush**$_{\text{opt}}$. The **flush**$_{\text{opt}}$ instruction is guaranteed to take effect (the contents of the given location reach the persistent memory) when a following **sfence** instruction is executed.

dTML$_{\text{Px86}}$ is designed to make use of **flush**$_{\text{opt}}$ instructions for efficiency. However, this introduces new verification challenges. Namely, **flush**$_{\text{opt}}$ instructions are difficult to reason about and, in fact, some earlier logics [58] only provided partial support for **flush**$_{\text{opt}}$ instructions, requiring a program with **flush**$_{\text{opt}}$ instructions to be transformed into a program with

[1] Instructions such as **flush**, **flush**$_{\text{opt}}$ and **sfence** actually apply to cache lines instead of locations. However, as in [8], for brevity we make the assumption that each cache line only holds one location, eliminating the need to reason about other locations on the same cache line.

flush instructions only. This transformation technique was known to be incomplete [58]. Full support for $\textbf{flush}_{\text{opt}}$ was only provided after development of the view-based semantics of Px86 (which we call Px86_{view}) [12] and a corresponding Owicki–Gries logic [8]. Our proofs for $\text{dTML}_{\text{Px86}}$ represent the first large-scale proofs of correctness for a realistic program that uses $\textbf{flush}_{\text{opt}}$ instructions.

We aim to achieve full operational proofs of correctness, therefore we build on the aforementioned Px86_{view} semantics [12] and Owicki–Gries logic [8]. However, using this logic directly in our current work is not possible for two reasons.

(1) Like prior works on verifying Px86 programs [12, 58], Bila et al. [8] have only focussed on reasoning about the behaviour *upto the first crash* of the program. To fully establish correctness of $\text{dTML}_{\text{Px86}}$, it is critical to also reason about the program after restarting the system.
(2) The assertions of PIEROGI defined by Bila et al. [8] are inadequate for reasoning about certain phenomena that occur in $\text{dTML}_{\text{Px86}}$. In particular, we must often reason about memory patterns by considering the order in which writes occur.

One of our contributions is an extension of the view-based semantics [12] as well as the associated logic [8] to enable reasoning about program recovery (after a crash), as well as new assertions that enable reasoning about the last writes to a location.

Our correctness proof of $\text{dTML}_{\text{Px86}}$ uses *forward simulation* to establish a refinement with respect to an abstract operational specification (called dTMS2 [5]). This, to our knowledge, is the first operational proof of refinement for the Px86. Other works have used refinement to verify durable linearisability directly under the *declarative* Px86 model [24, 56]. Unlike our work, these prior works are not accompanied by any mechanisation. Dalvandi and Dongol [16] have considered operational refinement proofs of transactional memory algorithms under the RC11 memory model with full mechanisation in Isabelle/HOL. These proofs have a different set of complexities (due to relaxed and release-acquire accesses), but do not require consideration of durability or recovery as we do in $\text{dTML}_{\text{Px86}}$.

1.2 Contributions

This paper comprises the following main contributions.

(1) We develop a durable transactional memory $\text{dTML}_{\text{Px86}}$ that guarantees durable opacity under Px86_{view} (and hence Px86). As mentioned above, $\text{dTML}_{\text{Px86}}$ makes use of $\textbf{flush}_{\text{opt}}$ instructions for improved efficiency, which increases the verification challenge.
(2) We develop a extension of the Px86_{view} semantics to enable operational reasoning about the behaviour of program *after* a crash, i.e., the recovery and subsequent execution. This is coupled with an extended Owicki–Gries logic that is also capable of reasoning about recovery steps.
(3) To take advantage of our operational reasoning technique, we apply a simulation-based proof to show correctness of $\text{dTML}_{\text{Px86}}$ by refinement. The proof proceeds via a long-established technique of establishing a forward simulation between the implementation and an abstract specification [5, 18, 22]. In the context of transactional memory, we prove that $\text{dTML}_{\text{Px86}}$ is a refinement of an operational model, dTMS2 [5], whose traces are guaranteed to be durably opaque.
(4) We mechanise our entire development in Isabelle/HOL, ranging from the semantics, logic (including soundness of the atomic Hoare triples), and all proofs pertaining to $\text{dTML}_{\text{Px86}}$, including proofs of the invariant and simulation.

1.3 Supplementary material

The Isabelle/HOL development accompanying this paper is available at [7].

1.4 Overview

This paper is organised as follows. In Sect. refsec:motivation, we provide some background and further motivation for our work, and in Sect. 3, we recap durable opacity as well as an operational model that guarantees durable opacity. In Sect. 4, we present a view-based operational model for Px86, including our extensions that model recovery after a crash. We present our extended $\mathrm{dTML}_{\mathrm{Px86}}$ algorithm in Sect. 5. We present the Owicki–Gries proof technique (further extended to cope with recovery) and the invariants of $\mathrm{dTML}_{\mathrm{Px86}}$ in Sect. 6. In Sect. 7 we present the durable opacity proof of $\mathrm{dTML}_{\mathrm{Px86}}$ and in Sect. 8 we discuss related work.

2 Background and motivation

In this section, we provide some basic high-level background and general motivation for our work.

2.1 Px86 semantics

To illustrate the behaviours of different persistent memory instructions, we use three examples (see Fig. 1) by Raad et al. [59], which demonstrate the behaviour of $\mathbf{flush}_{\mathrm{opt}}$ instructions. The assertion at the end of each program (indicated by $\lightning$) expresses *persistent invariant* [8], i.e., the persistent memory state if the corresponding program crashes.

The program in Fig. 1a first writes the value 1 to location x, then issues an optimised flush instruction to x. Finally, it writes the value 1 to location y. During its execution, both values 0 and 1 possible values for both locations x and y in the persistent memory. This is because $\mathbf{flush}_{\mathrm{opt}}\ x$ by itself does not guarantee that x is persisted before $\mathbf{store}\ y\ 1$ is executed. In fact, after executing $\mathbf{store}\ y\ 1$, it may be the case that y may be set to 1 in persistent memory *before* x is set to 1.

To prevent potential reorderings between optimised flushes and later instructions, one can use the **sfence** (store fence) instruction as mentioned above. Other options would be using (RMW) instructions such as a compare-and-swap (CAS) or fetch-and-add (FAA). As illustrated in Fig. 1b adding an **sfence** instruction before $\mathbf{store}\ y\ 1$ prevents the $\mathbf{flush}_{\mathrm{opt}}\ x$ from being reordered after it. Thus, if the persistent value of y is 1, then $\mathbf{store}\ y\ 1$ must have been executed, and hence **sfence** must have also been executed, which means that the persistent value of x is 1.

The program in Fig. 1c constitutes a message passing example. As in TSO, loading the value 1 for y (stored in register r) in the second thread, indicates that the store of value 1 at y from the first thread has been already evicted from its local store buffer. Since the store of value 1 to x precedes the store to y in the first thread, this means that the write to x has also been evicted from the first thread's store buffer and therefore is visible to the second thread. The $\mathbf{flush}_{\mathrm{opt}}\ x$ instruction in the second thread cannot be reordered before the preceding load. Hence, when the $\mathbf{flush}_{\mathrm{opt}}\ x$ is executed, the value of x seen by the second thread must be 1. Moreover, after the **sfence** is executed, we can be sure that the value of x in persistent

store x 1;
flush$_{\text{opt}}$ x;
store y 1;
(a)
	store x 1;
	flush$_{\text{opt}}$ x;
	sfence;
	store y 1;
	(b)
		store x 1;
		store y 1; ‖ $r :=$ **load** y;
		if $(r{=}1)$
		  flush$_{\text{opt}}$ x;
		  sfence;
		  store z 1;
		(c)

$\lightning$: $x, y \in \{0, 1\}$	$\lightning$: $y{=}1 \Rightarrow x{=}1$	$\lightning$: $z{=}1 \Rightarrow x{=}1$

Fig. 1 Example Px86 programs by Raad et al. [59] where the assertion $\lightning$ defines the possible persisted values during the execution. In all examples x, y, z are distinct locations with initial value 0, and r is a (thread-local) register.

memory is 1. Thus, if the persistent value of z is 1 (meaning that the store to z has been executed), then the persistent value of x is also 1.

2.2 Implementation challenges under Px86

Transactional memory (TM) aims to simplify concurrent programming by executing operations (loads, stores) within a transaction with an *illusion of atomicity*. That is, all changes to data inside a transaction are performed as if they were a single operation. Transactions also execute in an all-or-nothing manner—either all operations occur (i.e., the corresponding transaction *commits*), or none occur (i.e., the corresponding transaction *aborts*). We aim to develop a TM algorithm that ensures *durable opacity* [5], which we discuss in §3.

There are two main challenges when developing durable TM algorithms under weak memory models such as Px86.

(1) The first challenge concerns thread *synchronisation*. In a weak memory context, a read of a shared location may return a *stale value*, i.e., a value that is not the location's last written value. To address this, we must use instructions with strong ordering guarantees (e.g., **CAS**) at key points within dTML$_{\text{Px86}}$ to prevent transactions from reading stale values.

(2) The second challenge concerns *durability*. Without correct placement of explicit flush instructions and the careful design of a recovery mechanism, there is no guarantee of correctness after a system crash. To tackle this, we must strategically position **flush**$_{\text{opt}}$ and **sfence** instructions in a way that does not compromise the algorithm's efficiency. We must also design a recovery process that enables the state to be reset to a consistent state after a crash.

3 Durable opacity

Opacity has been extensively covered in the literature [2, 3, 17, 21, 33, 49], while the formal definition of durable opacity may be found in [5, 6]. We provide these formal definitions in Sect. C, and explain the key concepts here through example (Sect. 3.1). Formally, we only require an operational characterisation of durable opacity called dTMS2 [5, 6], which we present as an input/output automaton in Sect. 3.2. dTMS2 has been used in prior proofs of durable opacity [5, 6, 23] including recent model checking encodings under Px86 [61].

Note that in this paper, for simplicity, we conflate threads and transactions, i.e., each thread is assumed to execute at most one transaction. This restriction can easily be lifted, but at the cost of additional notational overhead [16], whereby we explictly track the transaction executed by each thread in a special state variable. In the following sections, we often use the terms *thread* and *transaction* interchangeably.

3.1 Opacity and durable opacity

The discussion and example below is adapted from our earlier work [23].

Correctness conditions for TM are defined in terms of *histories* of externally visible events, which are the external calls (invocations) and returns (responses) of TM operations. Typically, we have a pair of events for operations `TMBegin`, `TMRead`, `TMWrite` and `TMCommit`, noting that an operation call may return with an abort.

A *concurrent history* comprises an interleaving of (external) events from the different operations executed by different transactions. Each history is assumed to be *well formed*, i.e., the history, when restricted to a single transaction starts with a `TMBegin`, possibly followed by a number of `TMRead` and `TMWrite` operations, possibly followed by a `TMCommit` operation (see Fig. 2). Moreover, each operation executed by a transaction must have responded before the next operation is invoked.

A transaction is *complete* in a history if it has responded with `TMCommit(ok)` or an `abort` event, and once completed, the transaction must not execute any further operations. However, a transaction within a history may not be complete, i.e., may be a *live* transaction.

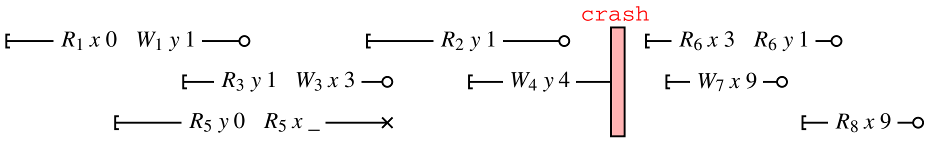
TM implementations are typically designed to be *serialisable*, i.e., there is a total order of *committed transactions* that is consistent with a sequential history. The TM implementations of interest in this paper in fact guarantee *strict serialisability*, which means that the total order of operations must additionally respect the real-time order, i.e., if transaction t_1 commits before transaction t_2 starts, then t_1 must serialise before t_2. Concurrent (i.e., overlapping) transactions may, however, be serialised in any order. TM implementations also typically provide a semantics for live and aborted transactions. A well-studied condition here is *opacity* [33], which ensures that there exists a total order across *all* transactions so that the committed transactions are strictly serialised and the aborted transactions are consistent with the serialisation order.

While the above provides semantics for transaction consistency, under NVM, we also require a further guarantee of *failure atomicity*. To this end, we follow the notion of *durable opacity* [5], where all transactions committed before a crash are persistent (after the crash), and in addition, the effects of any partially executed transactions are generally not visible after the crash. This concept is similar to that of durable linearisability [38], for concurrent objects.

A *durable concurrent history* is a concurrent history interleaved with `crash` events. A durable concurrent history is *well formed* iff the history with `crash` events removed is well formed and, moreover, no transaction that started before the crash continues executing after the crash.

Durable opacity, defined over durable concurrent histories, simply requires that the given history is opaque after all crash events are removed. Note that this means that any live transactions before a crash are aborted, and the writes of any committed transactions are persisted, i.e., are not lost after crash.

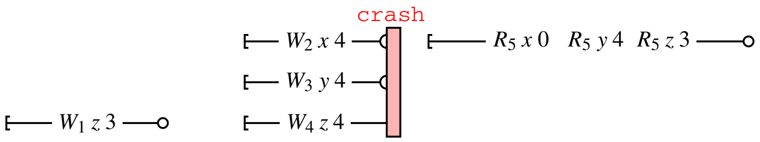
Example 1 (Dongol and Le-Papin [23]) Consider the history given below, where we elide the response events as well as the `TMBegin` / `TMCommit` operations, focussing instead on the allowable order of the transactions t_1-t_9. We use $R_i\ x\ v$ to denote a completed read operation by transaction t_i on variable x returning value v. (Similarly $W_i\ x\ v$.) We use $R_i\ x\ \_$ to denote a TMRead operation that has been invoked by t_i but not returned. All transactions except for transactions t_4 and t_5 are committed. Transaction t_4 is a live transaction that is interrupted by a crash, and transaction t_5 is an aborted complete transaction.



To show that the history above is durably opaque, we must remove the crash events, and show that the remaining history is opaque. Here, we must find a total order among *all* (including live and aborted) transactions so that the values returned by the read operations are consistent with the memory semantics w.r.t. the committed transactions. This total order must respect the real-time order of transactions, e.g., t_1 and t_2 may not be reordered. Assuming all variables are initialised to 0, an ordering that satisfies these constraints is: $t_5 \prec t_1 \prec t_3 \prec t_2 \prec t_4 \prec t_6 \prec t_7 \prec t_8$. Other orders are possible, however, for example, t_1 cannot occur before t_5 even though t_5 aborts (if it did, $R_5\ y\ 0$ would be inconsistent with the memory semantics).

One caveat of durable opacity pertains to transactions that have already invoked (but not returned from) TMCommit when a crash occurs. When removing crash events from the history, such transactions may either be treated as a committed transaction, or a live (and hence aborted) transaction [5].

Example 2 Consider the history given below, which comprises committed transactions t_1 and t_5 and live transactions t_2, t_3 and t_4 that are interrupted by a crash. We assume that both t_2 and t_3 have started committing when crash occurs, but t_3 has not.

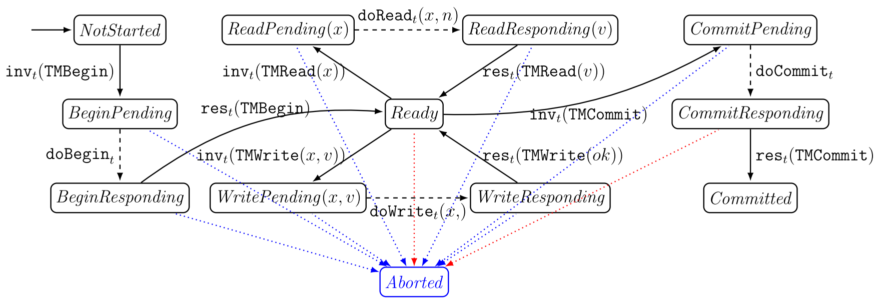


The history is durably opaque, e.g., $t_1 \prec t_2 \prec t_3 \prec t_4 \prec t_5$ is a valid total order, where we treat t_2 as an aborted transaction, but t_3 as a committed transaction. Transaction t_4 can only be considered as a live (and hence aborted) transaction.

3.2 The dTMS2 operational specification

While Sect. 3.1 provides a pedagogical overview of durable opacity, the formal aspects needed for this paper are provided by an operational specification, dTMS2 [5]. dTMS2 extends the TMS2 operational model [20] with a crash-recovery operation (Crash). The transitions of dTMS2 are given in Fig. 2. TMS2 has been shown to imply opacity [50], while dTMS2 has been shown to imply *durable opacity* [5]. Thus every history of dTMS2 is guaranteed to be durably opaque. Later in Sect. 6, we will show that dTML$_{\text{Px86}}$ is a refinement of dTMS2, and hence also guaranteed to satisfy durable opacity.

The memory of dTMS2 is modelled as a sequence of mappings from locations to values $L \in (\text{LOC} \to \text{VAL})^*$. We refer to each such mapping as a memory snapshot. Interestingly,



```
doBegin$_t$
pre : $pc_t = BeginPending \wedge t \neq syst$
eff : $pc_t = BeginResponding$;
      $beginIdx_t = |L| - 1$;

doCommit$_t$
pre : $pc_t = CommitPending \wedge t \neq syst \wedge$
      $((wrSet_t = \emptyset \wedge \exists n.\ validIdx(t,\ n)) \vee rdSet_t \subseteq last(L))$
eff : $pc_t = CommitResponding$;
      if $wrSet_t \neq \emptyset$ then
         $L := L +\!\!+ \langle last(L) \oplus wrSet_t \rangle$;

Crash
pre : $True$
eff : $\lambda t \in \mathrm{TID}.$ if $pc_t \notin \{NotStarted, Aborted, Committed\}$
            then $Aborted$ else $pc_t$;
      $L := \langle last(L) \rangle$;
```

```
doWrite$_t(x,v)$
pre : $pc_t = WritePending(x,v) \wedge t \neq syst$
eff : $pc_t := WriteResponding$;
      $wrSet_t := wrSet_t \oplus \{x \to v\}$;

doRead$_t(x,n)$
pre : $pc_t = ReadPending(x) \wedge t \neq syst \wedge$
      $(x \in \mathbf{dom}(wrSet_t) \vee validIdx(t,\ n))$
eff : if $x \in \mathbf{dom}(wrSet_t)$ then
         $v := wrSet_t(x)$;
         $pc_t := ReadResponding(v)$;
      else
         $v := L(n)(x)$;
         $rdSet_t := rdSet_t \oplus \{x \to v\}$;
         $pc_t := ReadResponding(v)$;
```

Fig. 2 Transitions of dTMS2 for a transaction t, where $validIdx(t, n) = beginIdx_t \leq n < |L| \wedge rdSet_t \subseteq L(n)$ and $\oplus$ denotes a functional override. For example $f \oplus \{x \to v\} = \lambda y.$ **if** $y = x$ **then** v *else* $f(y)$. Blue arrows represent possible transitions for either an abort or crash. Red arrows represent transitions that are possible for a crash only. Dashed arrows (e.g., `doBegin`$_t$) represent internal transitions.

as in other works on refinement-based proofs of durability [18], there is no need distinguish between volatile and persistent memory state in the abstract specification, and the entire state is considered to be persistent. Like dTML$_{\mathrm{Px86}}$, dTMS2 supports operations (`TMBegin`), read (`TMRead`), write (`TMWrite`) and commit (`TMCommit`).

Writing transactions of dTMS2 assume a deferred update policy, i.e., each transaction t maintains a write set ($wrSet_t$) that records the values that t has written during its execution (see `doWrite`$_t(x, v)$ in Fig. 2). The memory is updated by writing back the elements of the $wrSet_t$ to memory when the transaction commits (`TMCommit`). In particular, when a writing transaction t commits (see `doCommit`$_t$ in Fig. 2), t creates a new memory snapshot by applying its write set to final memory in L, then appending the resulting memory snapshot to L.

To ensure read consistency, the reads performed from memory are recorded in a local read set ($rdSet_t$) for each transaction t (see the **else** case of `doRead`$_t(x)$ in Fig. 2). To judge consistency, the largest memory index is stored in $beginIdx_t$, recording the earliest memory snapshot against which t can serialise (see `doBegin`$_t$ in Fig. 2). A read-only transaction must be validated w.r.t. *some* memory snapshot indexed at or after $beginIdx_t$ (see `doRead`$_t(x, n)$ in Fig. 2). Validation only succeeds (see *validIdx*) against a memory snapshot $L(n)$ if each read in the given read set is consistent with $L(n)$. A read in a writing transaction is similar except that the validity check must be w.r.t. the *last* memory snapshot when the location being read is not in the transaction's write set. If the location being read is in the transaction's write set, then the value in the write set is returned.

$$
\begin{array}{ll}
v,u\in \textsc{Val} \triangleq \mathbb{N} \quad x,y,\ldots\in\textsc{Loc} \quad o\in\textsc{Dobj} \quad f\in \mathrm{F} & \\
a,b,\ldots\in\textsc{Reg} \quad t\in\textsc{Tid}\triangleq\mathbb{N} \quad i,j,k,\ldots\in\textsc{Lab} & \\
\hat{a},\hat{b},\ldots\in\textsc{AuxVar} & \hat{e}\in\textsc{AuxExp} ::= v \mid \hat{a} \mid \hat{e}{+}\hat{e} \mid \cdots \\
\quad e\in\textsc{Exp} ::= v \mid a \mid e{+}e \mid \cdots & \textsc{BExp} ::= \text{boolean-valued } \textsc{Exp} \\
\quad \alpha\in\textsc{ASt} ::= \mathbf{skip} \mid a:=e \mid a:=\mathbf{load}\,x \mid \mathbf{store}\ x\ e & \\
\qquad\qquad \mid a:=\mathbf{CAS}\ x\ e\ e \mid \mathbf{mfence} \mid \mathbf{flush}\ x \mid \mathbf{flush_{opt}}\ x \mid \mathbf{sfence} \mid o.f & \\
\quad ls\in\textsc{LSt} ::= \alpha\ \mathbf{goto}\ j \mid \mathbf{if}\ B\ \mathbf{goto}\ j\ \mathbf{else\ to}\ k \mid \langle\alpha\ \mathbf{goto}\ j, \hat{a}:=\hat{e}\rangle & \\
\quad \Pi\in\textsc{Prog}\triangleq\textsc{Tid}\times\textsc{Lab}\to\textsc{LSt} & pc\in\textsc{PC}\triangleq\textsc{Tid}\to\textsc{Lab}
\end{array}
$$

Fig. 3 Programming language syntax.

As shown in Fig. 2, following the notion of a canonical automata [52], each ‘do’ transition is internal, and is preceded and succeeded by a corresponding external invocation and response transition. A transaction can abort (indicated by pc value *Aborted*) after invocation, but before responding. It can crash by transitioning to *Aborted* from any state after starting, but before it has committed or aborted.

4 View-based Px86 model

This paper builds on the *view-based model* for Px86$_{view}$ proposed by Cho et al. [12], which has been shown to be equivalent to Px86 [59]. Px86$_{view}$ abstractly captures underlying architectural complexities in terms of timestamps. To support the modelling and verification of our TM implementation, we extend Px86$_{view}$ as follows:

(1) We add a new CRASH transition to model a system-wide crash. This is needed because in contrast to prior work [12, 58], we wish to allow reasoning about our TM execution even after a crash/recovery event takes place.
(2) We introduce a syntax and semantics for high-level durably linearisable [38] objects.

In the following section, we provide a description of the Px86$_{view}$ programming language and semantics, emphasising on our extensions.

4.1 Programming language

The syntax of our language is given in Fig. 3, which is the syntax from prior work [8, 12] extended with high-level method calls.

Atomic statements (in ASt) may be a no-op (**skip**), a local assignment ($a := e$), a load of a shared location ($a :=$ **load** x), a store to a shared location (**store** x e), an atomic compare-and-swap ($a :=$ **CAS** x e_1 e_2), a memory fence (**mfence**), a flush instruction (**flush** x), an optimised flush instruction (**flush**$_{\text{opt}}$ x), a store fence (**sfence**) or a call to an atomic method f of object o ($o.f$).

A labelled statement LSt is either:

(1) a statement of the form α **goto** j, comprising an atomic statement α to be executed and the label j of the next statement;
(2) a conditional statement of the form **if** B **goto** j **else to** k, which facilitates branching, directing execution to label j if B holds and to k, otherwise; and
(3) a statement incorporating an auxiliary update, denoted as $\langle\alpha$ **goto** $j, \hat{a} := \hat{e}\rangle$. An auxiliary update behaves like α **goto** j, but additionally updates the value of the auxiliary variable $\hat{a}$ with the auxiliary expression $\hat{e}$ within the same atomic step.

Following [8], a program Π is represented as a function that maps pairs of the form $(t, i \in \text{TID} \times \text{LAB})$ to *labelled statements* in LST, representing the next statement to be executed.

Control flow within each thread is tracked by a *program counter function*, pc, which records the program counter of each thread. The initial label of each thread is a designated label ι (in LAB). During a program's execution, the pc value of a thread changes according to Π and at the end of the thread's execution, pc is assigned to a designated value $\zeta \in \text{LAB}$.

Example 3 (Program) The program Fig. 1a, assuming that the executing thread has id 1, is given as follows:

$$\Pi \triangleq \left\{ \begin{array}{l} (1, \iota) \mapsto \textbf{store}\ x\ 1\ \textbf{goto}\ 2, \\ (1, 2) \mapsto \textbf{flush}_{\text{opt}}\ x\ \textbf{goto}\ 3, \\ (1, 3) \mapsto \textbf{store}\ y\ 1\ \textbf{goto}\ \zeta \end{array} \right\}$$

4.2 The Px86$_{view}$ semantics

Our operational semantics is based on earlier work by Cho et al. [12]. A selection of transition rules of the semantics is given in Fig. 4.

A state is modelled by a tuple $\sigma = \langle pc, rec, \mathbb{T}, M, G \rangle$.

- $pc : \text{TID} \to \text{LAB}$ maps each thread to the next instruction to be executed.
- $rec : bool$ is a flag that indicates when a recovery process is in progress. In the event of a crash, rec is set to true to indicate that an implementation-specific recovery process is about to start its execution. We assume that after the recovery process completes, rec is reset to false.
- $\mathbb{T} : \text{TID} \to \text{THREAD}$ maps each thread to its current thread state, where THREAD is a record of *thread views* (see below) and local register store ($\mathsf{regs} : \text{REG} \to \text{VAL}$).
- $M \in \text{MEMORY}$ is a list of *messages* modelling the current memory. The first message of each memory is a store $CM : \text{LOC} \to \text{VAL}$, and the subsequent messages have the form $\langle \text{LOC} := \text{VAL} \rangle$. Initially, we assume $M = \langle CM \rangle$, where $CM(x) = 0$ for all $x \in \text{LOC}$.
- $G : \text{AUXVAR} \to \text{VAL}$ records the current values of auxiliary variable.

We denote the components of state σ as $\sigma.\mathbb{T}$, $\sigma.M$, etc. We refer to the indices of a memory list as *timestamps*. For $ts > 0$, the location and a value of a message m are denoted as $m.\mathsf{loc}$ and $m.\mathsf{val}$, respectively. The length of the memory list M is denoted as $|M|$. We say that a message with timestamp ts_1 and location x is not overwritten from timestamp ts_2's perspective if the following holds: $\forall ts \in (ts_1, ts_2].\ M[ts].\mathsf{loc} \neq x$. We denote the above as $x \notin M(ts_1..ts_2]$. Furthermore, we use $\sqcup$ to obtain the maximum among timestamps (i.e. $ts_1 \sqcup ts_2 = \max(ts_1, ts_2)$).

The views of a thread state THREAD comprises the following components.

- $\mathsf{coh} : \text{LOC} \to \mathbb{N}$, modelling the *coherence view*, which is used to determine the last write to the given location seen by the thread. In combination with $\mathsf{v_{rNew}}$ below, coh determines the range of observable values by t for a given location.
- $\mathsf{v_{rNew}} : \mathbb{N}$, modelling the latest timestamp among all timestamps seen by the thread.
- $\mathsf{v_{pReady}} : \mathbb{N}$, used to ensure that **load**, **sfence**, **mfence** and **CAS** instructions are ordered w.r.t. subsequent **flush**$_{\text{opt}}$ instructions.
- $\mathsf{v_{pAsync}} : \text{LOC} \to \mathbb{N}$, modelling the *asynchronous view*, which is used to determine values to be persistent after the execution of an **sfence**.
- $\mathsf{v_{pCommit}} : \text{LOC} \to \mathbb{N}$ modelling the *persistent view*, which is used to determine the set of values of a given location in persistent memory.

(ASSIGN)
$$\frac{\begin{array}{c}\alpha = a := e\\ v = T.\mathsf{regs}(e)\\ T' = T[\mathsf{regs}(a) \mapsto v]\end{array}}{\langle T, M\rangle \xrightarrow{\alpha} \langle T', M\rangle}$$

(STORE)
$$\frac{\begin{array}{c}\alpha = \textbf{store}\ x\ e\\ v = T.\mathsf{regs}(e)\\ M' = M \mathbin{+\!\!+} [\langle x := v\rangle]\\ T' = T[\mathsf{coh}(x) \mapsto |M|]\end{array}}{\langle T, M\rangle \xrightarrow{\alpha} \langle T', M'\rangle}$$

(LOAD-INTERNAL)
$$\frac{\begin{array}{c}\alpha = a := \textbf{load}\, x\\ M[ts] \equiv \langle x := v\rangle\\ T.\mathsf{coh}(x) = ts\\ T' = T[\mathsf{regs}(a) \mapsto v]\end{array}}{\langle T, M\rangle \xrightarrow{\alpha} \langle T', M\rangle}$$

(LOAD-EXTERNAL)
$$\frac{\begin{array}{c}\alpha = a := \textbf{load}\, x\\ M[ts] \equiv \langle x := v\rangle\\ T.\mathsf{coh}(x) < ts\\ x \notin M(ts..T.\mathsf{v}_{\mathsf{rNew}}]\end{array} \qquad T' = T\begin{bmatrix}\mathsf{regs}(a) \mapsto v,\\ \mathsf{coh}(x) \mapsto ts,\\ \mathsf{v}_{\mathsf{rNew}} \mapsto_{\sqcup} ts,\\ \mathsf{v}_{\mathsf{pReady}} \mapsto_{\sqcup} ts\end{bmatrix}}{\langle T, M\rangle \xrightarrow{\alpha} \langle T', M\rangle}$$

(SFENCE)
$$\frac{\begin{array}{c}\alpha = \textbf{sfence}\\ T' = T\begin{bmatrix}\mathsf{v}_{\mathsf{pReady}} \mapsto_{\sqcup} \bigsqcup_{x} T.\mathsf{coh}(x),\\ \mathsf{v}_{\mathsf{pCommit}} \mapsto_{\sqcup} T.\mathsf{v}_{\mathsf{pAsync}}\end{bmatrix}\end{array}}{\langle T, M\rangle \xrightarrow{\alpha} \langle T', M\rangle}$$

(CAS-SUCCESS)
$$\frac{\begin{array}{c}\alpha = a := \textbf{CAS}\ x\ e_1\ e_2\\ v_1 = T.\mathsf{regs}(e_1)\\ v_2 = T.\mathsf{regs}(e_2)\\ M[ts] = \langle x := v_1\rangle\\ x \notin M(ts..|M|]\\ M' = M \mathbin{+\!\!+} [\langle x := v_2\rangle]\end{array} \qquad T' = T\begin{bmatrix}\mathsf{regs}(a) \mapsto \mathsf{true},\\ \mathsf{coh}(x) \mapsto |M|,\\ \mathsf{v}_{\mathsf{rNew}} \mapsto |M|,\\ \mathsf{v}_{\mathsf{pCommit}} \mapsto |M|,\\ \mathsf{v}_{\mathsf{pReady}} \mapsto |M|\end{bmatrix}}{\langle T, M\rangle \xrightarrow{\alpha} \langle T', M'\rangle}$$

(CAS-FAIL-INTERNAL)
$$\frac{\begin{array}{c}\alpha = a := \textbf{CAS}\ x\ e_1\ e_2\\ M[ts] = \langle x := v\rangle\\ T.\mathsf{coh}(x) = ts\\ x \in M(t..|M|] \lor v \neq T.\mathsf{regs}(e_1)\\ T' = T\left[\mathsf{regs}(a) \mapsto \mathsf{false}\right]\end{array}}{\langle T, M\rangle \xrightarrow{\alpha} \langle T', M\rangle}$$

(CAS-FAIL-EXTERNAL)
$$\frac{\begin{array}{c}\alpha = a := \textbf{CAS}\ x\ e_1\ e_2\\ M[ts] = \langle x := v\rangle\\ T.\mathsf{coh}(x) < ts\\ (x \in M(t..|M|] \lor v \neq T.\mathsf{regs}(e_1))\end{array} \qquad T' = T\begin{bmatrix}\mathsf{regs}(a) \mapsto \mathsf{false},\\ \mathsf{coh}(x) \mapsto t,\\ \mathsf{v}_{\mathsf{rNew}} \mapsto_{\sqcup} t,\\ \mathsf{v}_{\mathsf{pReady}} \mapsto_{\sqcup} t\end{bmatrix}}{\langle T, M\rangle \xrightarrow{\alpha} \langle T', M\rangle}$$

(FLUSHOPT)
$$\frac{\begin{array}{c}\alpha = \textbf{flush}_{\text{opt}}\ x\\ T' = T[\mathsf{v}_{\mathsf{pAsync}}(x) \mapsto_{\sqcup} T.\mathsf{coh}(x) \sqcup T.\mathsf{v}_{\mathsf{pReady}}]\end{array}}{\langle T, M\rangle \xrightarrow{\alpha} \langle T', M\rangle}$$

(CRASH)
$$\frac{\begin{array}{c}T.\mathsf{coh} = (\lambda x.0) \qquad T.\mathsf{v}_{\mathsf{rNew}} = 0 \qquad T.\mathsf{v}_{\mathsf{pReady}} = 0\\ T.\mathsf{v}_{\mathsf{pAsync}} = (\lambda x.0) \qquad T.\mathsf{v}_{\mathsf{pCommit}} = (\lambda x.0)\end{array}}{\mathbb{T} \rightarrow (\lambda t \in \textsc{Tid}.T)}$$

(PROGRAM-NORMAL)
$$\frac{\begin{array}{c}pc(t) = i \qquad \Pi(t, i) = \alpha\ \textbf{goto}\ j\\ \langle \mathbb{T}(t), M\rangle \xrightarrow{\alpha} \langle T', M'\rangle\\ pc' = pc[t \mapsto j] \qquad \mathbb{T}' = \mathbb{T}[t \mapsto T']\end{array}}{\langle pc, \mathsf{false}, \mathbb{T}, M, G\rangle \Rightarrow_{\Pi} \langle pc', \mathsf{false}, \mathbb{T}', M', G\rangle}$$

(PROGRAM-IF)
$$\frac{\begin{array}{c}pc(t) = i\\ \Pi(t, i) = \textbf{if}\ B\ \textbf{then}\quad \textbf{goto}\ j\quad \textbf{else to}\ k\\ pc' = pc\left[t \mapsto \begin{cases} j & \mathbb{T}(t).\mathsf{regs}(B) = \mathsf{true}\\ k & \mathbb{T}(t).\mathsf{regs}(B) = \mathsf{false}\end{cases}\right]\end{array}}{\langle pc, \mathsf{false}, \mathbb{T}, M, G\rangle \Rightarrow_{\Pi} \langle pc', \mathsf{false}, \mathbb{T}, M, G\rangle}$$

(PROGRAM-GHOST)
$$\frac{\begin{array}{c}pc(t) = i \qquad \Pi(t, i) = \langle \alpha\ \textbf{goto}\ j, \hat{a} := \hat{e}\rangle\\ \langle \mathbb{T}(t), M\rangle \xrightarrow{\alpha} \langle T', M'\rangle\\ pc' = pc[t \mapsto j] \qquad \mathbb{T}' = \mathbb{T}[t \mapsto T']\\ G' = G[\hat{a} \mapsto G(\hat{e})]\end{array}}{\langle pc, \mathsf{false}, \mathbb{T}, M, G\rangle \Rightarrow_{\Pi} \langle pc', \mathsf{false}, \mathbb{T}', M', G'\rangle}$$

(PROGRAM-CRASH)
$$\frac{\begin{array}{c}\sigma = \langle pc, \mathsf{false}, \mathbb{T}, M, G\rangle \qquad \mathbb{T} \rightarrow \mathbb{T}'\\ pc' = pc[syst \mapsto Rec_{pending}]\\ \forall x \in \textsc{Loc}.\ CM(x) \in [x]^{\mathsf{P}}(\sigma)\end{array}}{\langle pc, \mathsf{false}, \mathbb{T}, M, G\rangle \Rightarrow_{\Pi} \langle pc', \mathsf{true}, \mathbb{T}', \langle CM\rangle, G\rangle}$$

Fig. 4 Sample of transition rules of Px86$_{view}$ for a program Π.

We assume that all the registers and views are initialised to 0.

As shown in Fig. 4, execution of a **store** $x\ v$ instruction adds a message $\langle x := v\rangle$ to the memory list and updates the coherence view. A $r :=$ **load** x instruction either reads from an earlier write performed by the same thread (LOAD- INTERNAL) or from a write performed by another thread (LOAD- EXTERNAL), which update different thread view components. If the read happens to read the first message of the memory returns $M[0](x)$, otherwise it returns $M[ts].\mathsf{val}$ (assuming $M[ts].\mathsf{loc}$). We capture both scenarios using the notation $M[ts] \equiv \langle x := v\rangle$.

The **CAS** instruction is modelled by two transition rules (CAS- SUCCESS and CAS- FAILURE). The CAS- SUCCESS transition (for $a :=$ **CAS** $x\ e_1\ e_2$) takes place when the value of register e_1 is equal to the last write at x (the last memory message with location x). In this case, a message $\langle x := v_2\rangle$, where v_2 is the value of register e_2, is appended in the end of the memory list and register a is assigned true. The CAS- FAILURE transition takes place when the last write at x does not have the value v_1. In this case, register a is assigned false. The effect of the CAS- FAILURE transition is a equivalent to the effect of a **load** instruction on location x.

A more detailed description of the views of a thread is given in §A, while an example execution is given below.

Example 4 (Program execution) Fig. 5 illustrates how the *view* components of a thread state $\mathbb{T}(t)$ change when t executes a program.

(1) Initially, the memory $\sigma.M$ only includes the *initial message*, and all the components of t's view state point to timestamp 0 (i.e., the initial message).
(2) After the execution of **store** x 1 the message $\langle x := 1\rangle$ is added to the memory. The store transition causes the coherence view of t for x (i.e., $\sigma.\mathbb{T}(t).\mathsf{coh}(x)$) to become 1.
(3) Execution of $\textbf{flush}_{\mathsf{opt}}$ x causes $\sigma.\mathbb{T}(t).\mathsf{v}_{\mathsf{pAsync}}(x)$ to point to memory index 1. Thus, after executing a subsequent **sfence**, $x := 1$ will be guaranteed to have been persisted (after step 4 below).
(4) The **sfence** instruction causes the $\sigma.\mathbb{T}(t).\mathsf{v}_{\mathsf{pCommit}}(x)$ view for t to point to the same message as the $\sigma.\mathbb{T}(t).\mathsf{v}_{\mathsf{pAsync}}(x)$ view, indicating that $x := 1$ is now persisted.
(5) Finally, execution of **store** y 1 adds to memory M the message $\langle y := 1\rangle$ at index 2. All the views in $\mathbb{T}(t)$ remain the same apart from the coherence view of y (i.e., $\sigma.\mathbb{T}(t).\mathsf{coh}(y)$).

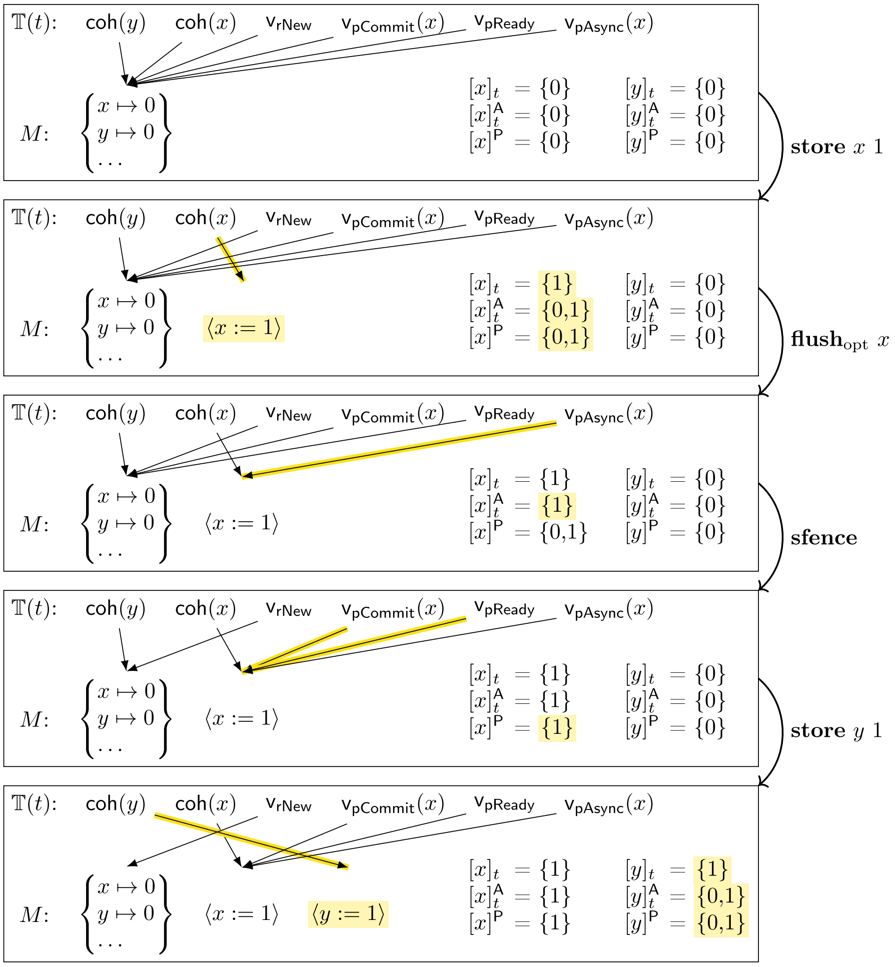
4.2.1 Modelling crashes and recovery

In contrast to prior works [8, 12], which only modelled execution upto the first crash, we provide explicit mechanisms to enable reasoning about crashes and the subsequent recovery operation. We introduce a CRASH transition that creates a new initial message and resets the views of each thread.

Specifically, the memory component of the state, $\sigma.M$, satisfies CM immediately after a crash in state σ if for every $x \in \textsc{Loc}$, there exists some ts such that $\sigma.M[ts] \equiv \langle x := CM(x)\rangle$ and $x \notin \sigma.M(ts..\bigsqcup_t \sigma.\mathbb{T}(t).\mathsf{v}_{\mathsf{pCommit}}(x)]$. To formalise this, we first define the set of possible persistent timestamps for location x in σ:

$$\mathsf{TS}^{\mathsf{P}}(\sigma, x) \triangleq \left\{ ts \mid \mathsf{MemLoc}(x, ts, \sigma.M) = x \wedge x \notin \sigma.M(ts..\bigsqcup_t \sigma.\mathbb{T}(t).\mathsf{v}_{\mathsf{pCommit}}(x)] \right\}$$

where $\mathsf{MemLoc}(x, t, M) \triangleq \textbf{if}\ (t = 0)\ \textbf{then}\ x\ \textbf{else}\ M[t].\mathsf{loc}$. The set of timestamps $\mathsf{TS}^{\mathsf{P}}(\sigma, x)$ represent the set of timestamps of messages that have been *persisted* in state σ, and thus their



$\mathbb{T}(t)$:
coh(y)
coh(x)
v$_{\mathsf{rNew}}$
v$_{\mathsf{pCommit}}(x)$
v$_{\mathsf{pReady}}$
v$_{\mathsf{pAsync}}(x)$
M:
$x \mapsto 0$
$y \mapsto 0$
...
$[x]_t = \{0\}$
$[x]^{\mathsf{A}}_t = \{0\}$
$[x]^{\mathsf{P}} = \{0\}$
$[y]_t = \{0\}$
$[y]^{\mathsf{A}}_t = \{0\}$
$[y]^{\mathsf{P}} = \{0\}$
store x 1
$\langle x := 1\rangle$
$[x]_t = \{1\}$
$[x]^{\mathsf{A}}_t = \{0,1\}$
$[x]^{\mathsf{P}} = \{0,1\}$
flush$_{\text{opt}}$ x
$[x]^{\mathsf{A}}_t = \{1\}$
sfence
$[x]^{\mathsf{P}} = \{1\}$
store y 1
$\langle y := 1\rangle$
$[y]_t = \{1\}$
$[y]^{\mathsf{A}}_t = \{0,1\}$
$[y]^{\mathsf{P}} = \{0,1\}$

Fig. 5 A depiction of a subset of the *current views*, the thread state ($\mathbb{T}(t)$), and Px86$_{view}$ memory list (M). The assertions over the thread state are explained in Example 8. The highlighted components of the state capture the effects of each instruction.

corresponding values can be read for location x if a crash occurs at this point of execution. This set corresponds to all the timestamps of the memory messages with location x that are not overwritten before maximum of each thread's v$_{\mathsf{pCommit}}$ view for location x (i.e., $\bigsqcup_t \sigma.\mathbb{T}(t).\mathsf{v}_{\mathsf{pCommit}}(x)$).

Example 5 Consider the program executed in Example 4. The set $\mathsf{TS}^{\mathsf{P}}(\sigma, x)$ changes as follows: $\mathsf{TS}^{\mathsf{P}}(\sigma, x) = \{\} \xrightarrow{\textbf{store } x\ 1} \mathsf{TS}^{\mathsf{P}}(\sigma, x) = \{0, 1\} \xrightarrow{\textbf{flush}_{\text{opt}}\ x} \mathsf{TS}^{\mathsf{P}}(\sigma, x) = \{0, 1\} \xrightarrow{\textbf{sfence}}$ $\mathsf{TS}^{\mathsf{P}}(\sigma, x) = \{1\} \xrightarrow{\textbf{store } y\ 1} \mathsf{TS}^{\mathsf{P}}(\sigma, x) = \{1\}$

The set of values corresponding to these timestamps is given by

$$[x]^{\mathsf{P}} \triangleq \lambda\sigma.\, \mathsf{Vals}(\mathsf{TS}^{\mathsf{P}}(\sigma, x), x, \sigma.M)$$

where $\mathsf{Vals}(TS, x, M) \triangleq \{\mathsf{MemVal}(x, t, M) \mid t \in TS\}$ returns the values at the given set of timestamps, assuming $\mathsf{MemVal}(x, t, M) \triangleq \textbf{if } (t = 0) \textbf{ then } M[0](x) \textbf{ else } M[t].\mathsf{val}$. We call the set $[x]^{\mathsf{P}}$ the *persistent view* of location x. The persistent view of any location x in $\textsc{Loc}$ is global (not specific to a thread) and captures the possible values of x in persistent memory. It constitutes one of the *view-based* expressions that we use to form assertions in the proof outlines of $\mathrm{Px86}_{view}$ programs [8]. We present other *view-based* expressions in Sect. 6.1.

We assume that recovery is executed by a unique system thread, *syst*, that is different from any program thread. Recovery is only enabled in state σ if $\sigma.rec$ holds. Moreover, we assume a special label, $Rec_{pending}$, which we assume is the label of the first recovery instruction. Upon completion of the recovery procedure, we assume that pc_{syst} is set to $Rec_{complete}$, and that there is a transition from this state to a state in which rec is set to false.

5 dTML$_{\mathrm{Px86}}$: a durable transaction mutex lock for Px86

In this section, we describe TML and the extensions required for durable opacity under Px86. An adaptation of TML that ensures durable opacity under the simpler PSC memory model (cf. [44]) has been presented in prior work [5]. In addition to assuming a more realistic memory model, unlike Bila et al. [5], our adapted algorithm dTML$_{\mathrm{Px86}}$, uses optimised flush instructions to increase performance [37], but at the cost of significantly increasing the verification challenge.

5.1 The dTML$_{\mathrm{Px86}}$ algorithm

Pseudocode for dTML$_{\mathrm{Px86}}$ is given in Fig. 6 as "fall-through" execution, which is notationally more convenient than our goto language (Sect. 4.1). Our Isabelle/HOL encoding uses the goto model (and hence is consistent with the language in Sect. 4.1).

In order to handle the weak behaviours introduced by Px86, we introduce several extensions to the original TML implementation [15]. Specifically, the lines highlighted blue ensure correct thread synchronisation under weak memory, while the lines highlighted green are required to ensure correctness under persistency. The variables highlighted grey are auxiliary. All the local variables apart from the auxiliary ones are modelled as registers. To distinguish them from global variables, we index the registers with the id of the transaction that they belong to. As before, we assume that thread identifiers coincide with the transaction identifiers. Moreover, for simplicity, line numbers for return statements are omitted. From now on, will use the term *internal* read for a read that a transaction performs to a location that the same transaction previously wrote, and *external* read for a read that a transaction performs to a location that has been written by another transaction.

We assume that all locations, the registers for every transaction, the global variable glb and the auxiliary variable recGlb, are initialised to zero. The auxiliary variable writer is initialised to *None*. We explain the behaviour of dTML$_{\mathrm{Px86}}$ in stages, starting with the basic algorithm.

5.1.1 The basic TML algorithm

TML performs writes in an *eager* manner, also known as *direct update*, i.e., it updates shared memory within the write operation itself. This is in contrast to lazy algorithms that store writes locally in a write set, and update shared memory at a later stage, e.g., during the commit

```
TMBegin
$Bp$ : do $\mathsf{loc}_t := \mathbf{load}\,\mathsf{glb}$;
$B1$ : until $even(\mathsf{loc}_t)$;
       return $ok$;

TMRead($x$)
$Rp$ : $r_t := \mathbf{load}\,x$;
$R1$ : if $even(\mathsf{loc}_t) \wedge \neg\mathsf{hasRead}_t$ then
$R2$ :    $\mathsf{hasRead}_t := \mathbf{CAS}\ \mathsf{glb}\ \mathsf{loc}_t\ \mathsf{loc}_t$;
$R3$ :    if $\mathsf{hasRead}_t$ then
             return $r_t$;
          else return $abort$;
$R4$ : $c_t := \mathbf{load}\,\mathtt{glb}$;
$R5$ : if $c_t = \mathsf{loc}_t$ then
          return $r_t$;
       else return $abort$;

TMCommit
$Cp$ : if $odd(\mathsf{loc}_t)$ then
$C1$ :    sfence;
$C2$ :    $log$.empty();
$C3$ :    $\langle$store glb $(\mathsf{loc}_t + 1)$,
          $\mathsf{writer} := None\rangle$
       return $commit$;

TMWrite($x, v$)
$Wp$ : if $even(\mathsf{loc}_t)$ then
$W1$ :    $\mathsf{hasWritten}_t := \mathbf{CAS}\ \mathsf{glb}\ \mathsf{loc}_t\ (\mathsf{loc}_t + 1)$;
$W2$ :    if $\mathsf{hasWritten}_t$ then
$W3$ :       $\langle \mathsf{loc}_t := \mathsf{loc}_t + 1, \mathsf{writer} := t\rangle$
          else return $aborted$
$W4$ : if $\neg log$.contains($x$) then
$W5$ :    $c_t := \mathbf{load}\,x$;
$W6$ :    $log$.update($x, c_t$);
$W7$ : store $x$ $v$;
$W8$ : $\mathbf{flush}_{\mathrm{opt}}$ $x$;
       return $ok$;

TMRecover
$Rec1$ :  while $\neg log$.isEmpty()
$Rec2$ :     $c_{syst} := log$.getKey();
$Rec3$ :     store $c_{syst}$ $log$.getVal($c_{syst}$);
$Rec4$ :     $\mathbf{flush}_{\mathrm{opt}}$ $c_{syst}$;
$Rec5$ :     sfence;
$Rec6$ :     $log$.update($c_{syst}, \perp$);
$Rec7$ :  $c_{syst} := \mathbf{load}\,\mathsf{glb}$;
$Rec8$ :  if $even(c_{syst})$ then
$Rec9$ :     $\langle$store glb $c_{syst} + 2$,
              $\mathsf{recGlb} := c_{syst} + 2\rangle$
$Rec10$ : else $\langle$store glb $(c_{syst} + 1)$,
              $\mathsf{recGlb} := c_{syst} + 1\rangle$
```

Fig. 6 Durable Transactional Mutex Lock.

operation. Additionally TML adopts a *strict* policy for transactional synchronisation: as soon as a transaction attempts to write to a variable, all other transactions running concurrently will be aborted when they invoke a read or a write operation. To enforce this synchronisation policy, TML uses a single *global versioned lock* [19], glb, and a local register loc_t to record a snapshot of glb at the beginning of the transaction t. A writing transaction is in progress iff the value of glb is odd.

A transaction t starts by calling TMBegin, then reading glb and storing the read value in the register loc_t (Bp). If the value of glb is odd, another writing transaction is in progress so t does not start. Instead, it reattempts to start by rereading glb.

Operation TMWrite(x, v) first checks whether loc_t is even (Wp). If not, then t must already be the writing transaction, and hence, it can proceed and update the value of the given location x to v ($W7$). If loc_t is even, it means that the current transaction is not yet a writing transaction, thus it attempts to become a writing transaction by performing a compare-and-swap (**CAS**) operation ($W1$). If this **CAS** succeeds, TMWrite becomes the writing transaction and increments loc_t ($W3$), making loc_t odd, then proceeds to update x to v ($W7$). In addition, at $W3$, the auxiliary variable writer is set to t. If the **CAS** at $W1$ fails, the transaction t aborts.

Operation TMRead(x) first reads the value at the given location x and stores it in the register r_t (Rp). The lines $R1$ to $R3$ are used to ensure weak-memory synchronisation under TSO and are explained below. At line $R4$, the operation reads the current value of glb. If this value is the same as loc_t, then either this transaction is the writing transaction, or no other transaction has performed any writes since this transaction started. In both cases the transaction returns the read value. If the test at $R5$ fails, then the transaction aborts.

Transaction t commits by first checking whether loc_t is odd (Cp). If so, it means that t is a writing transaction (and hence glb is odd), thus it makes glb even by incrementing glb and setting the auxiliary variable writer to *None*. If t is a read-only transaction (i.e., loc_t is even), it simply commits.

We now describe the necessary extensions for adapting TML to the persistent x86 setting. From now on, we assume that the underlying memory model is the persistent x86 and the instructions that are used correspond to the atomic statements of the Px86$_{view}$ programming language (see Sect. 4.1)

5.1.2 Correct synchronisation under Px86

Under Px86, in the presence of multiple writes to a location, a read may return a *stale value*, i.e., a value that is not the last written value. To ensure that a writing transaction serialises correctly, it must successfully perform a **CAS** at line $W1$, which guarantees that it reads the last written value of glb. However, in the standard TML and dTML algorithms [5, 15, 17] (which assume SC and PSC memory, respectively), this synchronisation is never performed by read-only transactions. Using approach in the Px86 setting is problematic since a read-only transaction may complete with a stale value of glb, without ever reading from the latest write to glb.

Example 6 Consider the program in Fig. 6 without lines $R1$–$R3$ (which have been introduced to address correctness under Px86). An execution of this program can reach a state with the following memory sequence:

$$\langle M_0, \langle \mathsf{glb} := 1\rangle, \langle x := 1\rangle, \langle \mathsf{glb} := 2\rangle, \langle \mathsf{glb} := 3\rangle, \langle x := 2\rangle, \langle \mathsf{glb} := 4\rangle\rangle$$

after executing two transactions t_1 and t_2, where t_1 writes 1 at location x and commits and afterwards t_2 writes 2 at location x and commits. Now suppose transaction t_3 starts, reads $\mathsf{glb} := 2$ (i.e. $\mathsf{loc}_t = 2$), allowing it to complete `TMBegin`, and then performs a `TMRead`(x) operation. The Px86 semantics allows it to read from the stale write $\langle x := 1\rangle$ (which has been written by transaction t_1), and then commit. Since $t_1 \prec t_2$ and $t_2 \prec t_3$, t_3 reading the value of x written by t_1, causes the generated history to violate the real-time ordering constraint of opacity.

To address this, we follow a similar approach to Dalvandi and Dongol [16] in the RC11 memory model,[2] and introduce a **CAS** in the `TMRead` operation ($R2$), mimicking a fetch-and-add-zero, to ensure that the last value of glb is read. If this **CAS** succeeds, the executing transaction can immediately return the read value, and if this **CAS** fails, the transaction can immediately abort ($R3$). Note that this **CAS** only needs to performed if the corresponding transaction has not previously performed a read or a write. Thus at line $R1$, we bypass $R2$ when loc_t is odd or $\mathsf{hasRead}_t$ holds. To see how the introduction of lines $R1 - R3$ addresses the issues, consider the following example.

Example 7 Consider the program in Fig. 6 (with lines $R1$–$R3$). Execution of this program can also reach the state in Example 6 after the execution of the transactions t_1 and t_2 described in Example 6. Once again, suppose transaction t starts, then reads $\mathsf{glb} := 2$ (i.e., $\mathsf{loc}_t = 2$),

[2] Note that although our solution to weak memory synchronisation is similar to the RC11 memory model [16], there are subtle differences in the way our solution guarantees correctness of reads. Unlike RC11 memory model which requires a "release" synchronisation on the read corresponding to Rp, in TSO, it is sufficient to perform a standard read.

allowing it to complete `TMBegin`. Suppose t then executes a $\texttt{TMRead}(x)$ operation reading the stale write $\langle x := 1\rangle$. However, now (unlike Example 6) t proceeds to line $R2$ and since loc_t is not the last written value of glb, the **CAS** fails, and thus t aborts.

5.1.3 Read-only transactions in Px86

Like Dalvandi and Dongol [16], we observe new behaviours of $\text{dTML}_{\text{Px86}}$ that would not be present under SC memory, but without violating durable opacity. In particular, a read-only transaction, t, is not immediately invalidated when glb is updated by another writing transaction, provided t continues to read from transactional locations that are consistent with a stale value of glb. This read-only transaction would be able to successfully commit if it *never* reads a value for x that is more recent than its copy of glb. In case a read-only transaction reads a value of a location x at Rp that is more recent than its local copy of glb, the load of glb at $R4$ would also read a more recent copy of glb and the transaction would subsequently abort.[3]

5.1.4 Ensuring durability

Durability of dTML under PSC has been studied in previous work [5]. The main idea there was to introduce a durably linearisable [38] persistent undo log that records the previous values of locations that have been overwritten by incomplete writing transactions. The log is reset to empty when the writing transaction commits. If a crash occurs when a incomplete writing transaction t is in flight, the subsequent recovery operation sets the state to the last consistent state by undoing the writes of t using the undo log. The recovery mechanism from the undo log is similar to this previous work [5], but we use $\textbf{flush}_{\text{opt}}$ and **sfence** instructions instead of **flush**.

As in earlier work [5], there is no need to explicitly persist glb. For transactions to successfully execute `TMBegin` after a crash, there is no necessity for transactions to read a particular value of glb at line Bp, as long as the read value is even. Lines $Rec8$–$Rec10$ of `TMRecover` ensure that there is at least one even value visible for glb after a system crash.

5.1.5 Alternative designs

While developing $\text{dTML}_{\text{Px86}}$, we considered several design alternatives. For instance, one option is to move the **CAS** instruction of line $R2$, to line Bp. In this way, a transaction t could have retried loading the most recent value of glb into loc_t until it succeeds before starting. This would have allowed the transaction to avoid aborting at a later stage. However, while this design may have resulted in fewer aborts, it would likely lead to a considerable increase in overall latency since transactions would be require to execute several **CAS** instructions within the `TMBegin` operation.

Another design alternative is to use a **flush** instruction instead of the $\textbf{flush}_{\text{opt}}$; **sfence** sequence in $Rec4$ and $Rec5$. Since the value of each location recorded in the log, is persisted sequentially and by only one thread, we expect the **flush** instruction in this case to be equally or more efficient than the current solution.

Both alternative designs would not affect significantly the verification effort.

[3] Note that this particular synchronisation property is much simpler to guarantee in Px86 than in the RC11 model [16], which requires careful management of release-acquire annotations.

5.2 $\text{dTML}_{\text{Px86}}$ model

We build a transition system model for $\text{dTML}_{\text{Px86}}$. In this model, we must clarify possible histories of the algorithm, which in turn requires us to clarify the invocation and response events. We assume that the algorithm is executed by a *most-general client* [22] that calls the operations of $\text{dTML}_{\text{Px86}}$.

5.2.1 *$\text{dTML}_{\text{Px86}}$* executions and histories

For each transaction t, we assume a *program counter*, pc_t, (initially *NotStarted*) that is used to model the control flow of transaction t. When t is in flight, but not executing any operation, we have $pc_t = \mathit{Ready}$. Similarly, $pc_t = \mathit{Aborted}$ and $pc_t = \mathit{Committed}$ iff t has aborted or committed, respectively. Otherwise pc_t is a line number corresponding to the instruction of the operation t is executing.

We assume each operation $op \in \{\texttt{TMBegin}, \texttt{TMRead}(x), \texttt{TMWrite}(x, v), \texttt{TMCommit}\}$ generates an event $inv_t(op)$ when op starts executing and $res_t(op)$, when op completes.

5.2.2 Ensuring well-formed histories

To ensure well-formedness of histories, we must ensure that transaction identifiers are not reused. Additionally, a live (i.e., in-flight) transaction before a crash must not continue its execution after the crash. To this end, we implicitly assume a *persistent transaction manager* that allocates new transaction identifiers. In our model, like earlier works [5] we use program counters to concisely characterise this assumption. First note that we assume program counter values of all threads except the system thread are unchanged after a CRASH transition (see Fig. 4), thus any transaction t with $pc_t = \mathit{NotStarted}$ can be executed after a crash. To ensure that in-flight transactions are not resumed, we assume that recovery starts by setting pc_t to *Aborted* for every transaction t such that $pc_t \notin \{\mathit{NotStarted}, \mathit{Aborted}, \mathit{Committed}\}$ (cf. `TMCrashRecovery` in Fig. 2).

5.2.3 Modelling log operations

The final source of complexity is the *durably linearisable* [38] log, *log*, which we model as a (persistent) mapping from locations to values. In our model, we use a sequential specification of *log* that does not enforce any *client-side memory synchronisation* (see [16, 66]) because the TML algorithm only allows a single writer at a time, and hence there is never any race on *log*. Moreover, because we assume that *log* is durably linearisable, the effect of each *log* operation is persisted before the operation returns, and hence its client (i.e., our $\text{dTML}_{\text{Px86}}$ algorithm) never accesses unpersisted *log* values. We assume that *log* supports the following operations.

- *log*.**isEmpty**() that returns true whenever the *log* is empty (i.e., all elements are mapped to $\perp$).
- *log*.**contains**(x) that returns true whenever the log contains x (i.e., x is not mapped to $\perp$).
- *log*.**contains**(x) that updates the logged location x to value v.
- *log*.**getKey**() that non-determinstically returns a location whose value is not $\perp$.
- *log*.**getVal**(x) that returns the value of x in *log*.

```
TMRead($x$)
$Pp : \{\mathsf{ready}_t\}$
$Rp : r_t := \mathbf{load}\, x;$
$P1 : \{$ $(\neg\mathsf{hasRead}_t \wedge \neg\mathsf{hasWritten}_t \wedge even(\mathsf{loc}_t) \wedge \mathsf{writer} \neq t \wedge (\mathsf{loc}_t = \overrightarrow{\mathsf{glb}} \Rightarrow r_t = \vec{x}))$
       $\vee$ $(\mathsf{hasRead}_t \wedge \neg\mathsf{hasWritten}_t \wedge even(\mathsf{loc}_t) \wedge \mathsf{writer} \neq t \wedge$
          $(\mathsf{loc}_t \in [\mathsf{glb}]_t \Rightarrow M[\mathsf{LE_{coh}}(\mathsf{glb}, t, x)] \equiv \langle x := r_t \rangle \wedge (\forall y.\ y \neq \mathsf{glb} \Rightarrow \mathsf{read_{pre}}(t, y))))$
       $\vee$ $(\mathsf{hasWritten}_t \wedge odd(\mathsf{loc}_t) \wedge \mathsf{writer} = t \wedge \mathsf{loc}_t = \overrightarrow{\mathsf{glb}} \wedge$
          $r_t = \vec{x} \wedge (\forall y.\ [y]_t = \{\vec{y}\}) \wedge (\forall y \in \mathbf{dom}(log).\ [y]_t^{\mathsf{A}} = \{\vec{y}\}))$ $\}$
$R1 :$ if $even(\mathsf{loc}_t) \wedge \neg\mathsf{hasRead}_t$ then
$P2 : \{\neg\mathsf{hasRead}_t \wedge \neg\mathsf{hasWritten}_t \wedge even(\mathsf{loc}_t) \wedge \mathsf{writer} \neq t \wedge (\mathsf{loc}_t = \overrightarrow{\mathsf{glb}} \Rightarrow r_t = \vec{x})\}$
$R2 :$    $\mathsf{hasRead}_t := \mathbf{CAS}\ \mathsf{glb}\ \mathsf{loc}_t\ \mathsf{loc}_t$
$P3 :$    $\{\mathsf{hasRead}_t \Rightarrow \mathsf{ready}_t\}$
$R3 :$    if $\mathsf{hasRead}_t$ then
             return $r_t$   $\{\mathsf{ready}_t\}$
          else return $abort$   $\{\mathsf{true}\}$
$P4 : \{$ $(\mathsf{hasRead}_t \wedge \neg\mathsf{hasWritten}_t \wedge even(\mathsf{loc}_t) \wedge \mathsf{writer} \neq t \wedge x \neq \mathsf{glb} \wedge$
          $(\mathsf{loc}_t \in [\mathsf{glb}]_t \Rightarrow M[\mathsf{LE_{coh}}(\mathsf{glb}, t, x)] \equiv \langle x := r_t \rangle \wedge (\forall y.\ y \neq \mathsf{glb} \Rightarrow \mathsf{read_{pre}}(t, y))))$
       $\vee$ $(\mathsf{hasWritten}_t \wedge odd(\mathsf{loc}_t) \wedge \mathsf{writer} = t \wedge \mathsf{loc}_t = \overrightarrow{\mathsf{glb}} \wedge x \neq \mathsf{glb} \wedge$
          $r_t = \vec{x} \wedge (\forall y.\ [y]_t = \{\vec{y}\}) \wedge (\forall y \in \mathbf{dom}(log).\ [y]_t^{\mathsf{A}} = \{\vec{y}\}))$ $\}$
$R4 : c_t := \mathbf{load}\,\mathsf{glb};$
$P5 : \{$ $(\mathsf{hasRead}_t \wedge \neg\mathsf{hasWritten}_t \wedge even(\mathsf{loc}_t) \wedge \mathsf{writer} \neq t \wedge x \neq \mathsf{glb} \wedge$
          $(c_t = \mathsf{loc}_t \Rightarrow M[\mathsf{LE_{coh}}(\mathsf{glb}, t, x)] \equiv \langle x := r_t \rangle \wedge (\forall y.\ y \neq \mathsf{glb} \Rightarrow \mathsf{read_{pre}}(t, y))))$
       $\vee$ $(\mathsf{hasWritten}_t \wedge odd(\mathsf{loc}_t) \wedge \mathsf{writer} = t \wedge x \neq \mathsf{glb} \wedge \mathsf{loc}_t = \overrightarrow{\mathsf{glb}} \wedge$
          $r_t = \vec{x} \wedge c_t = \mathsf{loc}_t \wedge (\forall y \in \mathbf{dom}(log).\ [y]_t^{\mathsf{A}} = \{\vec{y}\}))$ $\}$
$R5 :$ if $c_t = \mathsf{loc}_t$ then
          return $r_t$
       $\{\mathsf{ready}_t\}$
```

Fig. 7 TMRead annotation.

The log is stored in the G state component in Fig. 4 and updated according to SC semantics. An actual implementation of *log* may synchronise threads, e.g., with **mfence** operations, which affects the persistency and thread views of the variables of dTML$_{\mathrm{Px86}}$. Our proof makes no such assumptions about *log*, namely we assume the *weakest possible* ordering guarantees. Thus, an implementation of *log* that performs additional thread synchronisation would not affect soundness of our result.

6 Invariants of dTML$_{\mathrm{Px86}}$

This section describes the key invariants of dTML$_{\mathrm{Px86}}$ and mechanisms for proving their correctness. These will be used in the simulation proof in Sect. 7. Our work builds on the PIEROGI logic for Px86$_{view}$ [8], which uses view-based expressions derived from the *view* components of the thread state. We only require a subset of the PIEROGI assertions. However, we also introduce new view-based expressions simplify reasoning about dTML$_{\mathrm{Px86}}$ (see Sect. 6.1). This is combined with an Owicki–Gries style proof method to establish correctness of proof outlines (Sect. 6.2). However, unlike PIEROGI, because we additionally reason about the behaviour of a program after a crash, we slightly modify the interpretation of a persistent invariant as used in PIEROGI (see Sect. 6.2). PIEROGI requires that we establish a set of proof rules for atomic statements. We present a subset of these, including our new view-based expressions for dTML$_{\mathrm{Px86}}$ in §B. In Sect. 6.4, we present an example proof outline for the `TMRead` operation and finally, in Sect. 6.3, we present the persistent invariant.

6.1 View-based expressions

We first recap two key PIEROGI view-based expressions that are used in our proof.

The *thread view* expression, $[x]_t$, of a thread t for a location x captures the values that are visible to t for x. It indicates the values that can be read from t via the execution of a **load** or **CAS** instruction on x. The formal definition of $[x]_t$ is constructed by firstly specifying the set of timestamps of the visible to t memory messages with location x ($\mathsf{TS}_t(\sigma, x)$), and then by extracting the set of the values that correspond to those timestamps using Vals. We define:

$$[x]_t \triangleq \lambda\sigma.\, \mathsf{Vals}(\mathsf{TS}_t(\sigma, x), x, \sigma.M) \qquad \text{(thread view)}$$

where $\mathsf{TS}_t(\sigma, x) \triangleq \left\{ ts \,\middle|\, \begin{array}{l} \mathsf{MemLoc}(x, ts, \sigma.M) = x \wedge \\ \sigma.\mathbb{T}(t).\mathsf{coh}(x) \leq ts \wedge x \notin \sigma.M(ts..\sigma.\mathbb{T}(t).\mathsf{v_{rNew}}] \end{array} \right\}$.

Similarly, the *asynchronous view* expression, $[x]_t^{\mathsf{A}}$, of a thread t for a location x is thread-local and captures the values that can be persisted after the execution of an **sfence** instruction by t. This only depends on the view $\mathsf{v_{pAsync}}(x)$ of t, which potentially changes after a **flush**$_{\mathrm{opt}}$ on x by t. The formal definition of $[x]_t^{\mathsf{A}}$ is constructed by firstly specifying the set of timestamps of the asynchronous view of thread t for location x and state σ. Then, as before, we extract the set of values that correspond to those timestamps using Vals. We define:

$$[x]_t^{\mathsf{A}} \triangleq \lambda\sigma.\, \mathsf{Vals}(\mathsf{TS}_t^{\mathsf{A}}(\sigma, x), x, \sigma.M) \qquad \text{(asynchronous view)}$$

where $\mathsf{TS}_t^{\mathsf{A}}(\sigma, x) \triangleq \left\{ ts \mid \mathsf{MemLoc}(x, ts, \sigma.M) = x \wedge x \notin \sigma.M(ts..\sigma.\mathbb{T}(t).\mathsf{v_{pAsync}}(x)] \right\}$.

Example 8 Consider again the example execution in Fig. 5. This time we consider the assertions associated with each program state. Initially, views $[z]_t$, $[z]_t^{\mathsf{A}}$ and $[z]^{\mathsf{P}}$ for $z \in \{x, y\}$ all comprise the set $\{0\}$, meaning that the only value they can read is from the initial message.

(1) After execution of **store** x 1, we have $[x]_t = \{1\}$, since the coherence view changes, while $[x]_t^{\mathsf{A}} = [x]^{\mathsf{P}} = \{0, 1\}$ since these views can see the value for x in either the initial message or $\langle x := 1\rangle$. The view assertions on y are unchanged.
(2) After execution of **flush**$_{\mathrm{opt}}$ x, since $\mathsf{v_{pAsync}}(x)$ is updated, the value 0 is no longer visible to the asynchronous view, and hence $[x]_t^{\mathsf{A}} = \{1\}$. Note that the persistent memory may still see both 0 and 1 and hence $[x]^{\mathsf{P}} = \{0, 1\}$.
(3) Next **sfence** is executed, whereby the both $\mathsf{v_{pCommit}}(x)$ and $\mathsf{v_{pReady}}$ are updated, and this means that we have $[x]^{\mathsf{P}} = \{1\}$.
(4) Finally **store** y 1 is executed, which has a similar effect to the first step, but on y instead of x.

Next, we present an extension to PIEROGI that enable reasoning about written values before a given timestamp. The *last entry* views return the timestamp of the memory message with location equal to the given location and a timestamp less than or equal to the given limit.

$$\begin{aligned}
\mathsf{MemLastEntryLim}(x, t, M) &\triangleq \bigsqcup \{ts \mid \mathsf{MemLoc}(x, t, M) = x \wedge ts \leq t\} \\
\mathsf{LE}(x) &\triangleq \lambda\sigma.\, \mathsf{MemLastEntryLim}(x, |\sigma.M| - 1, \sigma.M) \\
\mathsf{LE_{coh}}(y, t, x) &\triangleq \lambda\sigma.\, \mathsf{MemLastEntryLim}(x, \mathsf{coh}_t(y)(\sigma), \sigma.M) \\
\vec{x} &\triangleq \lambda\sigma.\, \mathsf{MemVal}(x, \mathsf{LE}(x)(\sigma), \sigma.M)
\end{aligned}$$

$\mathsf{MemLastEntryLim}(x, t, M)$ returns the maximum timestamp of the memory messages with location x and timestamp less or equal to timestamp t, $\mathsf{LE}(x)$ returns the timestamp of the last memory message on location x, and $\mathsf{LE}_{\mathsf{coh}}(y, t, x)$ returns the timestamp of the last write to x before t's coherence view for y. The expression $\vec{x}$ returns the value of the last message of the memory with location x in the given state.

6.2 Owicki–Gries reasoning

In this section, we describe our Owicki–Gries style framework that we used to show that a proof outline is *valid*. Our framework follows PIEROGI [8], but we revise the notion of a persistent invariant to enable one to describe the execution of a program after a crash. In particular, given a multi-threaded program Π, in addition to the local correctness and global correctness checks, we also check that the persistent invariant is maintained by *all* program transitions, including those of the recovery operation. As such the persistent invariant can be used as an assumption when proving local correctness and global correctness. The use of a global invariant to simplify Owicki–Gries proofs is a well known technique [26].

We refer to the set of *assertions* (i.e. predicates over Px86$_{view}$ states) that use view-based expressions (§6.1) as an $\text{ASSERTION}_{\text{PV}}$. A *proof outline* is a tuple (in, ann, I, fin), where $in, fin \in \text{ASSERTION}_{\text{PV}}$ are the initial and final assertions, I is the persistent invariant and *ann* is an *annotation function* that models program annotations. Specifically, $ann \in \text{ANN} = \text{TID} \times \text{LAB} \to \text{ASSERTION}_{\text{PV}}$, associates each program point (t, i) with its associated assertion. We let *Recovery* denote the set of all statements of the recovery operation and *crash* be a statement corresponding to a CRASH transition.

Definition 1 (Valid proof outline) A proof outline (in, ann, fin, I) is *valid* for a program Π iff the following hold:

Initialisation. For all $t \in \text{TID}$, $in \Rightarrow I \wedge ann(t, \iota)$.
Finalisation. $I \wedge (\bigwedge_{t \in \text{TID}} ann(t, \zeta)) \Rightarrow fin$.
Local correctness. For all $t \in \text{TID}$ and $i \in \text{LAB}$, either:

- $\Pi(t, i) = \alpha$ **goto** j and $\{I \wedge ann(t, i)\}\ \alpha\ \{I \wedge ann(t, j)\}$; or
- $\Pi(t, i) =$ **if** B **goto** j **else to** k and both
  - $I \wedge ann(t, i) \wedge B \Rightarrow ann(t, j)$ and
  - $I \wedge ann(t, i) \wedge \neg B \Rightarrow ann(t, k)$ hold; or
- $\Pi(t, i) = \langle\alpha$ **goto** $j, \hat{a} := \hat{e}\rangle$ and $\{I \wedge ann(t, i)\}\ \alpha\ \{(I \wedge ann(t, j))[\hat{e}/\hat{a}]\}$.

Global Correctness. For all $t_1, t_2 \in \text{TID}$ such that $t_1 \neq t_2$ and $i_1, i_2 \in \text{LAB}$:

- if $\Pi(t_1, i_1) = \alpha$ **goto** j, then $\{I \wedge ann(t_2, i_2) \wedge ann(t_1, i_1)\}\ \alpha\ \{ann(t_2, i_2)\}$;
- if $\Pi(t_1, i_1) = \langle\alpha$ **goto** $j, \hat{a} := \hat{e}\rangle$, then $\{I \wedge ann(t_2, i_2) \wedge ann(t_1, i_1)\}\ \alpha$ $\{ann(t_2, i_2)[\hat{e}/\hat{a}]\}$.

Crash invariance. Both of the following hold:

- for all $\alpha \in Recovery$, $\{I\}\ \alpha\ \{I\}$
- $\{I\}\ crash\ \{I\}$

Initialisation (resp. Finalisation) ensures that the initial (resp. final) assertion of each thread holds in the initial (resp. final) state. Local correctness ensures the validity of the program annotation of each thread, while global correctness ensures the global correctness of the program annotation of each thread under the execution of other threads. In essence, the local correctness proof for a thread t checks for each atomic statement of t if its post-condition (given as annotation) can be established by its pre-condition (given as annotation). Similarly, the global correctness proof for a thread t checks that the pre-condition of each atomic statement of t is stable against the atomic statements of the other threads. Note that **if** B **goto** j **else to** k does not generate a global correctness proof obligation since B is an expression over thread-local variables, thus does not change the global state.

To show that a proof outline is valid (Definition 1) we use two types of rules: standard decomposition rules and rules for atomic statements.

6.2.1 Standard decomposition rules

The standard decomposition rules of Hoare logic such as weakening preconditions, strengthening postconditions, and decomposing conjunctions and disjunctions apply (see [8]).

6.2.2 Rules for atomic statements and correctness of view-based assertions

The proof rules that we use constitute all the rules of the PIEROGI framework [8] as well as some additional rules developed enable proofs of correctness for dTML$_{\mathrm{Px86}}$. All the proof rules used in this work have been mechanised and proved sound against our extensions to Px86$_{view}$ in Isabelle/HOL. Each rule captures the impact of the execution of atomic statements (discussed in §4.1) on assertions formed by view-based expressions (outlined in §6.1).

We have two general types of rules used to discharge local and global correctness proof obligations. Local correctness proof rules often describe how views are changed through the execution of a thread:

Example 9 Assuming that the statement in question is executed by thread t, the rule $\left\{[x]_t^{\mathsf{A}} = S\right\}$ $\textbf{flush}_{\text{opt}}\ x$ $\left\{[x]_t^{\mathsf{A}} \subseteq S\right\}$ states that the *asynchronous view* of x for thread t in the post-state is equal or a subset of its *asynchronous view* in the pre-state, after executing $\textbf{flush}_{\text{opt}}\ x$.

Global correctness proof rules are often used to show stability of assertions.

Example 10 Assuming that the statement in question is executed by thread t and $t \neq t'$, the rule $\left\{[y]_{t'} = S\right\}$ **sfence** $\left\{[y]_{t'} = S\right\}$ states that the *thread view* of any address y for any thread remains unchanged after the execution of **sfence**.

Other global correctness rules describe how the memory (and hence available values for a thread to observe change).

Example 11 Assuming that the statement in question is executed by thread t and $t \neq t'$, the rule $\left\{[x]_{t'} = S\right\}$ **store** $x\ v$ $\left\{[x]_{t'} = S \cup \{v\}\right\}$ states that the value v for x is available for thread t' to read after the execution of **store** $x\ v$.

A full set of rules for proving local and global correctness of view-based assertions is presented in Sect. B.

6.3 Persistent invariant of dTML$_{\mathrm{Px86}}$

To prove corrrectness of dTML$_{\mathrm{Px86}}$, we construct a multithreaded program $\Pi_{\mathrm{dTML_{Px86}}}$ based on the model introduced in Sect. 5.2. $\Pi_{\mathrm{dTML_{Px86}}}$ includes all dTML$_{\mathrm{Px86}}$ operations, invocation events, response events and the system crash event. With the exception of the system thread, which is only capable of executing the `TMRecover` operation, any thread t in TID is free to perform any number of operations (excluding the recovery operation) as long as the resulting execution history conforms to the control flow and well-formedness constraints.

In this section, we present the most important aspects of the persistent invariant, which comprises a collection of properties that the dTML$_{\text{Px86}}$ implementation guarantees in every program state. The corresponding proofs have been mechanised in Isabelle/HOL.

6.3.1 Memory properties

The first three properties describe memory patterns that occur during the execution of dTML$_{\text{Px86}}$. In each of the properties below, we assume that $i, j \in \mathbf{dom}(M)$ and that $i < j$.

Property 1 The values of glb are monotonically increasing within the memory sequence M, i.e.,

$$\forall v_i, v_j.\ M[i] \equiv \langle \mathsf{glb} := v_i \rangle \wedge M[j] \equiv \langle \mathsf{glb} := v_j \rangle \quad \Longrightarrow \quad v_i \le v_j$$

Property 1 is needed because unlike in prior work [5], the recovery process of dTML$_{\text{Px86}}$ does not reset glb to 0. This is actually necessary to avoid `TMRead` operations returning stale values (i.e., values that were in persistent memory, but subsequently modified) after a crash. The following example demonstrates this phenomenon.

Example 12 Consider the program in Fig. 6 that resets glb to zero (**store** glb 0) after $Rec6$ instead of executing lines $Rec7 - Rec10$. An execution of this program can reach a state with the following memory sequence:

$$\langle \{\mathsf{glb} \mapsto 2, x \mapsto 5, \_ \mapsto 0\}, \langle x := 3 \rangle, \langle \mathsf{glb} := 0 \rangle, \langle \mathsf{glb} := 1 \rangle, \langle y := 1 \rangle, \langle \mathsf{glb} := 2 \rangle \rangle$$

which is reached from the initial state after a

(1) A writing transaction updates x to 3 then commits (so $\mathsf{glb} = 2$),
(2) Another writing transaction writes updates x to 5 (so $log(x) = 3$),
(3) A crash occurs (resulting in the intial state above),
(4) The modified recovery operation described above executes (appending $\langle x := 3 \rangle$ then $\langle \mathsf{glb} := 0 \rangle$ to the memory),
(5) A third writing transaction that updates y to 1 commits successfully.

Now assume that another transaction t starts, then reads 2 for glb from the *initial message*, allowing it to complete `TMBegin`, then performs a `TMRead`(x) operation. In this case, according Px86 semantics the initial value of x (i.e., 5) is still observable at Rp. The test at $R1$ succeeds and the **CAS** instruction at $R2$ can still succeed, since the last value of glb is 2. As a result, t can successfully complete the `TMRead` operation and subsequently commit, violating durable opacity.

Property 2 If there exists a write between two writes to glb such that the value of glb is unchanged, then the location of any intermediate write between these two writes must be on glb, i.e.,

$$\forall v.\ M[i] \equiv \langle \mathsf{glb} := v \rangle \wedge M[j] \equiv \langle \mathsf{glb} := v \rangle \quad \Longrightarrow \quad \forall k \in [i, j].\ M[k].\mathsf{loc} = \mathsf{glb}$$

Property 2 holds since this memory pattern described by the antecedent only occurs when two or more transactions that have not yet executed a `TMRead` or `TMWrite` invoke `TMRead` operations and successfully execute their **CAS** instruction at $R2$. The first of these reading transactions introduces a write to glb that immediately follows either

(1) The initial message, or
(2) A write to glb by a writing transaction at $C3$, or
(3) A message added by the `TMRecover` process at $Rec9$ or $Rec10$.

The subsequent `TMRead` operations introduce writes to glb with unchanged values.

Property 3 Between a memory message on glb with even value and another memory message on a location different from glb, there exists a message with location on glb with odd value, i.e.,

$$i > 0 \wedge M[i].\mathsf{loc} = \mathsf{glb} \wedge even(M[i].\mathsf{val}) \wedge M[j].\mathsf{loc} \neq \mathsf{glb} \implies \exists k \in (i, j).\ M[k].\mathsf{loc} = \mathsf{glb} \wedge odd(M[k].\mathsf{val})$$

Property 3 describes a memory pattern that occurs when a transaction successfully performs a `TMWrite`. Note that excluding the initial message and the messages added from the recovery process, the only way that messages with a location different from glb are added to the memory is by executing $W7$. Prior to this, the writing transaction performs a successful **CAS** at $W1$. The execution of $W1$ adds a message to memory with location glb and odd value.

6.3.2 Coherence property for non-writing transactions

The next property uses $\mathsf{maxcoh}_t \triangleq \lambda\sigma.\ \bigsqcup_x (\sigma.\mathbb{T}(t)).\mathsf{coh}(x)$, which denotes the maximum coherence value for t across all locations and $\mathsf{vrnew}_t \triangleq \lambda\sigma.\ (\sigma.\mathbb{T}(t)).\mathsf{v}_{\mathsf{rNew}}$, which retrieves the value of $\mathsf{v}_{\mathsf{rNew}}$ for t. We let $\mathsf{Recovering} \triangleq \lambda\sigma.\ \sigma.rec = \mathsf{true}$.

Property 4 When a `TMRecover` process is not in progress, for any transaction that is not a writing transaction, the coherence view for all the locations in memory is less than or equal to its $\mathsf{v}_{\mathsf{rNew}}$ view, i.e.,

$$\forall t \in \textsc{Tid}.\ \neg\mathsf{Recovering} \wedge \mathsf{writer} \neq t \implies \mathsf{maxcoh}_t \leq \mathsf{vrnew}_t.$$

Property 4 holds because the only cases in which $\mathsf{coh}_t(x) > \mathsf{vrnew}_t$, is when t is executing a write on x or performing an internal read to x. Both cases are precluded for non-writing transactions.

6.3.3 Properties about tracked locations and log

We now describe a set of properties describing the memory locations that are tracked by Px86 and log. Note that we assume that all locations in $\textsc{Loc}$ different from glb can be transactionally written and read.

Property 5 The domain of log does not contain the location glb, i.e.,

$$\forall x \in \mathbf{dom}(log).\ x \neq \mathsf{glb}$$

Property 6 For all locations $x \neq \mathsf{glb}$ that is not in log, the persistent view includes only their last written value, i.e.,

$$\forall x \in \textsc{Loc}.\ x \neq \mathsf{glb} \wedge x \notin \mathbf{dom}(log) \implies [x]^{\mathsf{P}} = \{\vec{x}\}$$

6.3.4 Properties about glb and recGlb

Next we have three properties for glb and the auxiliary variable recGlb.

Property 7 In the presence of a writing transaction, last value of glb in the memory must be odd, i.e.,

$$\mathsf{writer} \neq \mathit{None} \implies \mathit{odd}(\overrightarrow{\mathsf{glb}})$$

Property 7 holds due to the successful execution of $W1$. Note that the implication does not hold in the other direction because, in our model, we reset the auxiliary variable writer to *None* during a crash, yet the last value of glb after a crash may be odd. One could have defined a stronger invariant: $\neg\mathsf{Recovering} \implies (\mathsf{writer} \neq \mathit{None} \Leftrightarrow \mathit{odd}(\overrightarrow{\mathsf{glb}}))$, however, we have not needed this strengthening in our proofs.

Property 8 With the exception of the initial message, the value of glb is greater than or equal to recGlb, i.e.,

$$\forall i \in \mathbf{dom}(M).\ 0 < i \wedge M[i].\mathsf{loc} = \mathsf{glb} \implies M[i].\mathsf{val} \geq \mathsf{recGlb}$$

Property 9 After a transaction t successfully executes `TMBegin`, the value of loc_t must be less than or equal to the last value of glb ($\overrightarrow{\mathsf{glb}}$). Moreover, after a successful `TMWrite` and/or `TMRead` operation has taken place (i.e. $\mathsf{hasRead}_t \vee \mathsf{hasWritten}_t$ holds), the value of recGlb is less than or equal to loc_t, i.e.

$$\forall t \in \mathrm{TID}.\ (pc_t \notin \{\mathit{NotStarted}, \mathit{Bp}, B1, B2, \mathit{Aborted}, \mathit{Committed}\} \implies \mathsf{loc}_t \leq \overrightarrow{\mathsf{glb}}) \wedge$$
$$(\mathsf{hasRead}_t \vee \mathsf{hasWritten}_t \implies \mathsf{recGlb} \leq \mathsf{loc}_t)$$

6.3.5 Properties about recovery

Finally, we have a set of properties about the state immediately after a crash (before recovery has begun) and after recovery has finished.

Property 10 When a `TMRecover` process is in progress, all the transactions are either *NotStarted*, *Aborted* or *Committed*, i.e.,

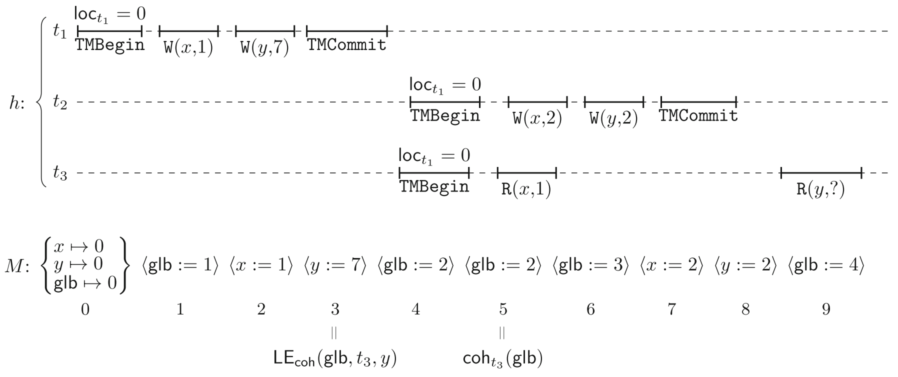
$$\mathsf{Recovering} \implies (\forall t.\ pc_t \in \{\mathit{NotStarted}, \mathit{Aborted}, \mathit{Committed}\})$$

Property 11 When a `TMRecover` process is not in progress (i.e., has completed), the value of glb in the initial message is less than the value of the auxiliary variable recGlb, which in turn is at most the final value of the value of recGlb. Moreover, the value of *even*(recGlb) is even, i.e.,

$$\neg\mathsf{Recovering} \implies M[0](\mathsf{glb}) < \mathsf{recGlb} \wedge \mathsf{recGlb} \leq \overrightarrow{\mathsf{glb}} \wedge \mathit{even}(\mathsf{recGlb})$$

6.4 dTML$_{\mathrm{Px86}}$ program annotation

We now enumerate the local properties of each thread by adding program annotations at each atomic step. The program annotation is formed by view-based expressions (see Sect. 6.1). As an example, we give the annotated proof outline of `TMRead` in Fig. 7. The assertions of dTML$_{\mathrm{Px86}}$ can be classified into three categories:



$\mathsf{loc}_{t_1} = 0$
t_1
TMBegin
W(x,1)
W(y,7)
TMCommit
h:
t_2
$\mathsf{loc}_{t_1} = 0$
TMBegin
W(x,2)
W(y,2)
TMCommit
t_3
$\mathsf{loc}_{t_1} = 0$
TMBegin
R(x,1)
R(y,?)
M: $\{x \mapsto 0, y \mapsto 0, \mathsf{glb} \mapsto 0\}$ ⟨glb := 1⟩ ⟨x := 1⟩ ⟨y := 7⟩ ⟨glb := 2⟩ ⟨glb := 2⟩ ⟨glb := 3⟩ ⟨x := 2⟩ ⟨y := 2⟩ ⟨glb := 4⟩
0 1 2 3 4 5 6 7 8 9
$\mathsf{LE}_{\mathsf{coh}}(\mathsf{glb}, t_3, y)$
$\mathsf{coh}_{t_3}(\mathsf{glb})$

Fig. 8 Example execution for read-only transactions.

(1) Transactions that have not yet performed a read or a write (green assertions),

(2) *Read-only* transactions (pink assertions), and

(3) *Writing* transactions (blue assertions).

The assertions highlighted yellow in Fig. 7 capture the effects of the preceding instruction.

We define an assertion ready_t, which holds when an in-flight transaction is in an idle state (i.e., not executing any transactional operation):

$$\begin{aligned}\mathsf{ready}_t = {} & \Big(\neg\mathsf{hasRead}_t \wedge \neg\mathsf{hasWritten}_t \wedge \mathit{even}(\mathsf{loc}_t) \wedge \mathsf{writer} \neq t \wedge \big(\mathsf{loc}_t = \overrightarrow{\mathsf{glb}} \implies \forall y. [y]_t = \{\vec{y}\}\big)\Big) \\ & \vee \big(\mathsf{hasRead}_t \wedge \neg\mathsf{hasWritten}_t \wedge \mathit{even}(\mathsf{loc}_t) \wedge \mathsf{writer} \neq t \wedge \big(\forall y.\ y \neq \mathsf{glb} \implies \mathsf{read}_{\mathsf{pre}}(t, y)\big)\big) \\ & \vee \left(\begin{array}{l}\mathsf{hasWritten}_t \wedge \mathit{odd}(\mathsf{loc}_t) \wedge \mathsf{writer} = t \wedge \mathsf{loc}_t = \overrightarrow{\mathsf{glb}} \wedge \\ (\forall y. [y]_t = \{\vec{y}\}) \wedge (\forall y \in \mathbf{dom}(\mathit{log}).\ [y]^{\mathsf{A}}_t = \{\vec{y}\})\end{array}\right)\end{aligned}$$

The first disjunct captures two local conditions of t: that the local snapshot of glb is even and the writer is not t, as well as a visibility guarantee that if t's the local snapshot of glb is consistent with the last write to glb, then the thread's view of each location y is the last write to y. The visibility guarantee ensures that the transaction t can be serialised after the last writing transaction in case t successfully performs its reads and commits.

The second disjunct covers read-only transactions as described in Sect. 5.1 using the predicate $\mathsf{read}_{\mathsf{pre}}(t, y)$ below. We let $\mathsf{coh}_t(x) \triangleq \lambda\sigma.\ (\sigma.\mathbb{T}(t)).\mathsf{coh}(x)$.

$$\mathsf{read}_{\mathsf{pre}}(t, y) = \mathsf{coh}_t(\mathsf{glb}) > 0 \wedge M[\mathsf{coh}_t(\mathsf{glb})] \equiv \langle \mathsf{glb} := \mathsf{loc}_t \rangle$$

Predicate $\mathsf{read}_{\mathsf{pre}}(t, y)$ is established a successful CAS- SUCCESS transition (see Fig. 4). Namely, the successful **CAS** transition at $R2$ in Fig. 7 shifts the coherence view of glb to the length of the memory in the pre-state, which is greater than zero since the memory includes always the initial message. Furthermore, the second conjuct of $\mathsf{read}_{\mathsf{pre}}$ holds because the successful **CAS** transition appends the message $\langle \mathsf{glb} := \mathsf{loc}_t \rangle$ to the end of the memory.

The third disjunct of ready is straightforward since a writing transaction takes the lock (by making glb odd). The only additional guarantee one required is that t's asynchronous view of each location in *log* is maximal. This guarantees that when t later performs an **sfence** at $C1$, all of the writes performed by t are persisted.

We discuss correctness of read-only transactions (pink assertions), which is the most challenging aspect of the proof. We use the example in Fig. 8 with an abstract history h

comprises three transactions t_1-t_3. Transactions t_1 and t_2 cannot be reordered due to the *real-time* order constraint of durable opacity (see Definition 4). Moreover, since the first read of transaction t_3 has returned 1 for x, the only valid sequential history corresponds to the ordering $(t_1 \prec t_3 \prec t_2)$. Thus, the second read in transaction t_3 must either return 7 for y, or abort.

In the implementation, we must identify the timestamp of the write that a read-only transaction reads from and does not lead to an abort. To this end, let

$$\mathsf{LE}_{\mathsf{coh}}(y, t, x) \triangleq \lambda\sigma.\, \mathsf{MemLastEntryLim}(x, \mathsf{coh}_t(y)(\sigma), \sigma.M)$$

where $\mathsf{LE}_{\mathsf{coh}}(y, t, x)$ returns the timestamp of the last write to x before t's coherence view for y. In the example in Fig. 8, we have $\mathsf{LE}_{\mathsf{coh}}(\mathsf{glb}, t_3, y) = 3$ since t_3's coherence view of glb is memory index 5, and the last write to y before index 5 is at index 3.

Provided that t_3 reads from the memory at index 3, the message at index 5 will still be observable to t_3. Therefore, it can read this message at $R4$ so that the check at $R5$ does not fail. We can prove that the second read of t_3 can only succeed if it reads from index 3 by contradiction.

Case 1: t_3 reads a message with timestamp greater than $\mathsf{LE}_{\mathsf{coh}}(\mathsf{glb}, t_3, y)$ *at Rp*. In Fig. 8, the only such message is at index 8. Using Property 3, in the post-state of Rp, there exists a timestamp ts between $\mathsf{coh}_{t_3}(\mathsf{glb})$ (i.e., 5 in our example) and $\mathsf{coh}_{t_3}(y)$ (i.e., 8 in our example), such that $M[ts] \equiv \langle \mathsf{glb} := v\rangle$ and $odd(v)$. In our example, $ts = 6$ and $v = 3$.

By Property 4, every observable timestamp for glb must be greater than or equal to ts (i.e. 6) and thus greater than $\mathsf{coh}_{t_3}(\mathsf{glb})$ (i.e., 5). Thus, t_3 must read a value for glb at $R4$ that is different from loc_{t_3}. By the third conjunct of the pink disjunct of ready_{t_3}, we have $even(\mathsf{loc}_{t_3})$. Moreover by Property 1, each value of glb after ts is at least v. Since $odd(v)$, we have $v \neq \mathsf{loc}_{t_3}$, thus t_3 cannot observe loc_{t_3} for glb.
Case 2: t_3 reads a message with timestamp less than $\mathsf{LE}_{\mathsf{coh}}(\mathsf{glb}, t_3, y)$ *at Rp*. In Fig. 8, such a message is the initial message (with timestamp 0). By Property 4, vrnew_t must be at least $\mathsf{coh}_{t_3}(\mathsf{glb})$ (i.e., timestamp 5). However, $\mathsf{LE}_{\mathsf{coh}}(\mathsf{glb}, t_3, y)$ overwrites this earlier message and hence the earlier timestamp is no longer visible to t_3.

The annotations for the remaining $\text{dTML}_{\text{Px86}}$ operations are given in Sect. B.1.

Theorem 1 () *The proof outline for $\text{dTML}_{\text{Px86}}$ is valid.*

7 Durable opacity via refinement

We now are now ready to prove durable opacity. The proof proceeds by showing refinement between $\text{dTML}_{\text{Px86}}$ and the dTMS2 operational specification (see Sect. 3.2). In particular, we establish a forward simulation (Definition 2) between the $\text{dTML}_{\text{Px86}}$ transition system and the dTMS2 specification. It is well known that a forward simulation is a sound proof technique for refinement. As in proofs of linearisability [22], refinement must guarantee trace inclusion, i.e., every externally observable behaviour of the concrete system (e.g., $\text{dTML}_{\text{Px86}}$) is an externally observable behaviour of the abstract system (e.g., dTMS2). The external steps (transitions) correspond to invocations and responses of transactional operations as well as the system crashes.

Definition 2 (Forward simulation) For an abstract system A and a concrete system C, a relation R between the states of A and C is a *forward simulation* iff each of the following holds.

Initialisation. For any initial state cs_0 of C, there exists an initial state as_0 of A such that $R(as_0, cs_0)$.
External step correspondence. For any external transition from cs to cs' in C and any state as of A such that $R(as, cs)$, there exists a corresponding external transition (performing the same action) from as to as' such that $R(as', cs')$.
Internal step correspondence. For any internal transition from cs to cs' of C and any state as of A such that $R(as, cs)$, either:

- $R(as, cs')$, or (stuttering step)
- There is an internal transition of A from as to as' such that $R(as', cs')$ (non-stuttering step)

The forward simulation relation (*simRel*) is split into two relations: a global relation, *globalRel* (see Sect. 7.1), and a transaction relation, *txnRel* (see Sect. 7.2). The global relation describes how the shared states of the two transition systems are related, while the transaction relation specifies the relation between the state of each transaction in the concrete and abstract transition systems. In particular, we have:

$$simRel(as, cs) \triangleq globalRel(as, cs) \wedge \forall t \in \textsc{Tid}.\ txnRel(as, cs, t)$$

where as and cs are states of dTMS2 and dTML, respectively.

To explain these relations, we start by identifying the *linearisation points* of the $\text{dTML}_{\text{Px86}}$ operations.

Operation `TMBegin` linearises at $B1$ provided loc_t is even. For transactions that have not performed any `TMRead` or `TMWrite`, the linearisation point of `TMRead` is a successful **CAS** at $R2$. For any other type of transaction, `TMRead` linearises at $R5$ provided the value read from glb is loc_t. Operation `TMWrite` linearises when the memory is updated at $W7$. Operation `TMCommit` has two linearisation points depending on whether the transaction has successfully executed a `TMWrite` operation. For a writing transaction (i.e., when loc_t is odd), `TMCommit` linearises at $C2$. Otherwise, `TMCommit` linearises at Cp.

Our simulation relation assumes the following auxiliary definitions. We let $intHalf(n) \triangleq \lfloor \frac{n}{2} \rfloor$, be integer division of n by 2.

$$\begin{aligned}
writes(as, cs) &\triangleq \textbf{if } cs.\mathsf{writer} = t \wedge pc_t \neq C3 \textbf{ then } as.\mathsf{wrSet}_t \textbf{ else } \emptyset \\
logicalGlb(cs) &\triangleq \textbf{if } cs.\mathsf{writer} = t \wedge pc_t = C3 \\
&\quad\ \textbf{then } \overrightarrow{\mathsf{glb}}(cs) - cs.\mathsf{recGlb} + 1 \textbf{ else } \overrightarrow{\mathsf{glb}}(cs) - cs.\mathsf{recGlb} \\
wrCount(cs) &\triangleq intHalf(logicalGlb(cs)) \\
inFlight(t, cs) &\triangleq t \in \textsc{Tid} \wedge pc_t \notin \{NotStarted, Aborted, Committed\}
\end{aligned}$$

The function *writes* returns the abstract wrSet_t of a writing transaction. Note that the abstract wrSet_t is empty after the writing transaction has cleared its log, and hence `TMCommit` linearises at $C2$. The function *logicalGlb* is used to determine the logical value of glb (since initialisation or the last recovery) and compensates for the fact that a committing writing transaction has linearised but not yet incremented glb at $C3$. The function $wrCount(cs)$ returns the number of committed writing transactions in the concrete state, taking into account the fact that each writing transaction updates glb twice. Finally, *inFlight* is used to determine whether the given transaction t in state cs is *live* (has been started but has not been committed or aborted).

7.1 Global relation

The global relation *globalRel* comprises conditions (1)-(4) below. The definition relies on $\mathsf{LE}(t, x) \triangleq \lambda\sigma.\, \mathsf{MemLastEntryLim}(x, t, \sigma.M)$, which returns the timestamp of the last write to x before timestamp t.

$$\neg\mathsf{Recovering} \implies (\forall x.\, x \neq \mathsf{glb} \implies \vec{x} = (\mathit{last}(as.L) \oplus \mathit{writes}(cs, as))(x) \quad (1)$$

$$\neg\mathsf{Recovering} \implies (\mathit{wrCount}(cs) = |as.L| - 1) \quad (2)$$

$$\forall x.\, x \neq \mathsf{glb} \implies \mathit{last}(as.L)(x) = \textbf{if } x \notin \textbf{dom}(cs.\mathit{log}) \; \textbf{then } \vec{x} \textbf{ else } (cs.\mathit{log})(x) \quad (3)$$

$$\forall i.\, \forall v.\, cs.M[i] \equiv \langle \mathsf{glb} := v\rangle \implies \forall x.\, \forall w.\, x \neq \mathsf{glb} \wedge cs.M[\mathsf{LE}(i, x)] \equiv \langle x := w\rangle \implies as.L[\mathit{intHalf}(v - cs.\mathsf{recGlb})](x) = w \quad (4)$$

Condition (1) states that, for each location x different from glb, the last written value for x in dTML$_{\mathrm{Px86}}$ is the value of x in the last memory snapshot of dTMS2 overwritten by the write set of an in-flight writing transaction (if there is any).
Condition (2) states that the number of memory snapshots in dTMS2 memory since initialisation or the last crash is equal to $\mathit{wrCount}(cs)$.
Condition (3) states that, for each location x different from glb, the value of x in the last memory snapshot of dTMS2 is the last written value for x in dTML$_{\mathrm{Px86}}$ whenever x is not in *log* and is the corresponding value in *log*, otherwise.
Condition (4) states that whenever dTML$_{\mathrm{Px86}}$'s memory index i contains a write to glb with value v, for any location x different from glb, the value of the last write to x before index i is precisely the value of x in the abstract memory snapshot indexed $\mathit{intHalf}(v - cs.\mathsf{recGlb})$.

7.2 Transaction relation

The transaction relation (txnRel) comprises conditions (5)-(10) given below, as well as a condition matching abstract and concrete program counters, return values, and validity of completing transactions, which we discuss later.

$$\forall t.\, \mathit{inFlight}(t, cs) \wedge \neg cs.\mathsf{hasWritten}_t \wedge \neg cs.\mathsf{hasRead}_t \implies as.\mathsf{rdSet}_t = \emptyset \quad (5)$$

$$\forall t.\, \mathit{inFlight}(t, cs) \wedge cs.\mathsf{hasRead}_t \implies as.\mathsf{rdSet}_t \neq \emptyset \quad (6)$$

$$\mathit{allt}.\, \mathit{inFlight}(t, cs) \wedge (\neg cs.\mathsf{hasWritten}_t \vee \mathit{even}(cs.\mathsf{loc}_t)) \implies as.\mathsf{wrSet}_t = \emptyset$$
$$\forall t.\, \mathit{inFlight}(t, cs) \wedge \mathit{odd}(cs.\mathsf{loc}_t) \wedge \quad (7)$$

$$cs.pc_t \notin \{Bp - B1, W4 - W7\} \implies as.\mathsf{wrSet}_t \neq \emptyset \quad (8)$$

$$\forall t.\, \mathit{inFlight}(t, cs) \wedge \mathsf{writer} = t \wedge cs.pc_t \notin \{W4 - W7\} \implies as.\mathsf{wrSet}_t \neq \emptyset \quad (9)$$

$$\forall t.\, \forall x \in \textbf{dom}(as.\mathsf{wrSet}_t).\, cs.\mathsf{writer} = t \implies (as.\mathsf{wrSet}_t)(x) = cs.\vec{x} \quad (10)$$

The first five conditions, relate the dTML$_{\text{Px86}}$ state of an *inFlight* transaction t with the wrSet_t and rdSet_t variables of the corresponding dTMS2 state. Condition (10) resolves *internal* reads and states that the write set (of the dTMS2 state) of a writing transaction t contains the last written value to that location by dTML$_{\text{Px86}}$ for each location in its domain.

We elide formal presentation of the final condition of *txnRel*, and instead provide a textual description of its remaining parts. We refer the interested reader to our mechanisation [7] for full details.

(1) *txnRel* maps dTML$_{\text{Px86}}$ program counter values to their dTMS2 counterparts, which also enables one to identify the linearisation points of dTML$_{\text{Px86}}$.

(2) *txnRel* ensures that the value returned by a dTML$_{\text{Px86}}$'s successful read on x ($\mathtt{TMRead}(x)$) in its linearisation point is the same as the value returned in the corresponding non-stuttering step of dTMS2. For this, the last condition leverages both the global relation and the `TMRead` annotation (Fig. 7). The way in which abstract and concrete values are matched differs for read-only and writing transactions. We give an overview of the proof for identifying the corresponding abstract and concrete values below.
Read-only transaction. The first read (i.e., $\mathtt{TMRead}(x)$) of a read-only transaction t succeeds if $cs.\mathsf{loc}_t$ obtains the maximum value of glb in cs. Otherwise, the **CAS** instruction at $R2$ fails. Based on the precondition of $R2$ ($P2$) (see Fig. 7), if the **CAS** instruction succeeds, we can deduce that $\overrightarrow{\mathsf{glb}}(cs) = cs.\mathsf{loc}_t$ and $cs.\mathsf{loc}_t$ is even. Additionally, we can infer that there is no message with a timestamp greater than or equal to the timestamp corresponding to the last write to glb in cs with a location different from glb. This is because, according to Property 3, if such a write existed, the value of the last write of glb would be odd. Therefore, the timestamp of the message read for x precedes the timestamp of the message of the last write to glb in cs and must have the form $\mathsf{LE}(i, x)$, where $cs.M[i] \equiv \langle \mathsf{glb} := cs.\mathsf{loc}_t \rangle$. By instantiating Condition (4), we can infer that the value read for x corresponds to the abstract value $as.L[\mathit{intHalf}(cs.\mathsf{loc}_t - cs.\mathsf{recGlb})](x)$.

For any subsequent read operation (i.e., $\mathtt{TMRead}(x)$) performed by a read-only transaction t, we can derive the index of the memory snapshot of dTMS2 that contains the returned write directly by the `TMRead` program annotation. Specifically, the `TMRead` program annotation (assertion $P5$ in Fig. 7) imposes that the only value that can be successfully returned by dTML$_{\text{Px86}}$ corresponds to the concrete memory message with timestamp $\mathsf{LE}_{\mathsf{coh}}(\mathsf{glb}, t, x)$. By expanding the definition of $\mathsf{LE}_{\mathsf{coh}}$, we obtain $\mathsf{LE}_{\mathsf{coh}}(\mathsf{glb}, t, x) = M[\mathsf{LE}(\mathsf{coh}_t(\mathsf{glb}), x)]$. Given this, and by using condition (4), we can determine that the index of the memory snapshot of dTMS2 containing this write is $as.L[\mathit{intHalf}(cs.\mathsf{loc}_t - cs.\mathsf{recGlb})]$.
Writing transaction. By the `TMRead` program annotation (assertion $P5$ in Fig. 7), a read operation on x (i.e., $\mathtt{TMRead}(x)$) of a writing transaction t can only return the last value written on x ($\vec{x}(cs)$). In case the read is *external*, by utilising condition (1), we can deduce that the corresponding abstract value is equal to $\mathit{last}(as.L)(x)$. In case the read is *internal* by condition (10) the corresponding abstract value is equal to $(as.\mathsf{wrSet}_t)(x)$.

(3) It ensures that the ordering (validity) constraints of dTMS2 are met. For this it guarantees that in the linearisation points of the dTML$_{\text{Px86}}$'s `TMRead` and `TMCommit` operations, we have:

$$as.\mathsf{rdSet}_t \subseteq as.L[\mathit{intHalf}(cs.\mathsf{loc}_t - cs.\mathsf{recGlb})] \wedge \\ as.\mathsf{beginIdx}_t \leq \mathit{intHalf}(cs.\mathsf{loc}_t - cs.\mathsf{recGlb}) \tag{11}$$

For a transaction t, the validity constraint of dTMS2 requires that if its read set ($as.\mathsf{rdSet}_t$) is not empty, it must be consistent with a memory snapshot indexed greater than or equal to $\mathsf{beginIdx}_t$ and less than the length of the abstract memory (as indicated in $\texttt{doRead}_t(x, n)$ and $\texttt{doCommit}_t$ in Fig. 2). By condition (11), condition (2), and property (9), an index satisfying these conditions exists and is equal to $intHalf(cs.\mathsf{loc}_t - cs.\mathsf{recGlb})$. In the case where t is a writing transaction ($\mathsf{wrSet}_t \neq \perp$), this index should correspond to the last element of the abstract memory (as defined in $\texttt{doCommit}_t$ in Fig. 2). The `TMCommit` annotation (see Appendix) specifies that at the linearisation point of the `TMCommit` operation for a writing transaction, $cs.\mathsf{loc}_t = \mathsf{glb}$. Combining this with condition (11), we can deduce that $as.\mathsf{rdSet}_t \subseteq as.L[intHalf(\vec{(}\mathsf{glb})(cs) - cs.\mathsf{recGlb})]$. According to condition (2), this corresponds to the last element of the abstract memory.

(4) It ensures that immediately after a crash, the length of the dTMS2 memory list is 1, the transaction that executes the `TMRecover` operation is $syst$ and the value of each location x that is read and cleared from the $cs.log$ in each recovery loop is equal to the corresponding value of x the memory snapshot of dTMS2.

Theorem 2 ( ) *simRel is a forward simulation is a between dTMS2 and dTML*$_{Px86}$.

7.3 Mechanisation

The refinement proof has been fully mechanised in Isabelle/HOL. This mechanisation builds on the PIEROGI framework [8]. This comprised three main steps:

(1) Encoding the necessary modifications to the Px86$_{view}$ model [12] to reflect the revised version presented in Sect. 4, then adapting a selection of the PIEROGI proof rules to the new context and proving the additional proof rules of PIEROGI (Sect. B).
(2) Proving validity of the persistent invariant of dTML$_{\text{Px86}}$ (Sect. 6.3) and the proof outline for dTML$_{\text{Px86}}$ (Theorem 1).
(3) Proving that the dTML$_{\text{Px86}}$ implementation refines the dTMS2 specification (Theorem 2). Specifically, we established the simulation relation for each step of the dTML$_{\text{Px86}}$ transition system, resulting in a total of 47 sub-proofs.

Steps (1) and (2) together took approximately 3 months of full-time work and step (3) required approximately 2 months.

The core development, including semantics, view-based assertions, and the soundness of proof rules, consists of approximately 10,000 lines of Isabelle/HOL code. With this foundation in place, the proof of the persistence invariant, and the validity of the proof outline for dTML$_{\text{Px86}}$ encompass approximately 20,000 lines of Isabelle/HOL code.

7.3.1 Lessons learnt

The initial step involves developing a persistent invariant and program annotations for dTML, which we expressed using PIEROGI [8] *view-based* expressions. While it was necessary to extend the PIEROGI expression syntax to account for memory patterns that occurred in dTML$_{\text{Px86}}$, a substantial subset of them remained applicable without modification. Thus, we believe that PIEROGI [8] can be widely utilised or easily extended to verify persistent memory algorithms in general. We extend the existing semantics [12] to model both crash and recovery. Additionally, we use an extended Owicki–Gries [54] logic that makes use of a persistent invariant, which is shown to hold for all program transitions, including the crash transition and the subsequent recovery process. These extensions can also be readily applied to model and reason about other programs, including after a crash.

The subsequent step comprises proving *forward simulation* [52] together with the dTML$_{\mathrm{Px86}}$ persistent invariant and program annotation, establishing refinement between dTML$_{\mathrm{Px86}}$ and dTMS2 [5] (which in turn guarantees durable opacity). Although, this was not used in the current proof, a strength of a simulation-based approach lies in its ability to enable hierarchical reasoning, e.g., if one was required to use intermediate model to establish both a forward and backward simulation [22, 49]. Our simulation relation is inspired by existing works [18], showing that established concepts and methods for verifying volatile algorithms provide a stepping stone to the more complex Px86 domain.

Each Px86$_{view}$ assertion that we use requires introduction of proof rules for this assertion for different atomic program statements (see §B), which must be proved sound against the operational semantics. Proving correctness of these rules can be challenging because it requires examination of the low-level operational semantics and their effect on the views of different system components. However, once soundness is established, they can be reused to validate proof outlines without extra effort. In particular, Isabelle/HOL can generate the required proof obligations with minimal interaction and then automatically identify the appropriate set of high-level proof rules needed to resolve each obligation using the integrated Sledgehammer tool [9].

In our proof (see [7]), there are several repeated patterns of unfoldings and theorem application. These could be automated through specific tactics (e.g., using Eisbach [53]). One could also make further improvements to the proof structure and modularisation (e.g., using locales [43]). However, we leave these aspects for future work.

8 Related work

Various works focus on adapting algorithms for the conventional volatile RAM model to non-volatile memory (NVM). FliT [70] is a C++ library that can be used for making any linearisable [34] data structure durable linearisable [38] by selectively flushing only writes that are subsequently read. NVTraverse [28] and Mirror [29] are able to translate automatically lock-free data structures into a durable data structures. In particular NVTraverse requires the given lock-free data structure to be in a traversal form while Mirror employs a shadowing data technique which requires maintaining two replicas of data, a persistent memory version which is updated first, and a volatile version which is updated second and is used for fast data access. In this work we are focussing on adapting a software transactional memory implementation to the persistency setting.

Our example implementation, dTML$_{\mathrm{Px86}}$ extends TML [15] with a persistent undo log, and associated modifications such as the introduction of a recovery operation. The undo log technique is used by several persistent STMs [11, 13, 46] as a means of achieving failure atomicity. An alternative technique comprises using a redo log [30, 32, 51, 62, 69]. Other persistent transactional memory algorithms rely on applying hardware modifications for achieving failure atomicity [40, 63, 67]

The literature includes numerous notions of correctness for *software transactional memory* many of which have been introduced as consistency conditions of database system transactions. As an example, strict serialisability [55], requires that all non-aborted completed transactions must be ordered to form a sequential history that is valid (i.e. respect the memory semantics) and respects the *real-time order* of the transactions. Although opacity can offer robust safety guarantees that render it suitable for transactional memory, it may be viewed as overly restrictive and needlessly complex to implement in TM systems. This is primarily due to its requirement that every live and abort transaction must be consistent with all prior committed transactions. To this end a number of correctness conditions have been suggested which aim to modify various aspects of opacity while preserving its essential safety guarantees such as elastic opacity [27], live opacity [25], virtual world consistency [35] and last-use opacity [65].

In the context of non-volatile memory, Raad et al. [57] proposed a notion of durable serialisability under weak memory which extends the concept of serialisability to the persistency setting. However, this correctness condition does not handle aborted transactions. Here we are focusing on durable opacity which was first introduced in by Bila et al. [5].

The PIEROGI logic, including the extensions developed in the current work, is proven sound against the Px86$_{view}$ semantics of Cho et al. [12]. Other operational semantics for persistent TSO include [1, 44, 59] and [60] which extend the model introduced in [59] to encompass non-temporal writes and reads and writes to a richer set of Intel-x86 memory types. In terms of program logics, apart from PIEROGI, POG [58] also addresses persistent memory programs. However, it is not mechanised and can not directly handle examples that involve **flush**$_{\text{opt}}$ instructions. Vindum and Birkedal [68] have recently developed a not architecture-specific concurrent separation logic for weak persistency. This logic is built upon the Perenial [10] and Iris logic framework [42] and has been mechanised in the Coq proof assistant.

Numerous works have aimed to simplify proofs of persistent memory programs (including persistent TM algorithms). Gorjiara et al. [31] have developed a notion of *robustness* for Px86, which holds if every post-crash execution of a program under Px86 is a post-crash execution under a strict persistency model. Here, strict persistency is defined in terms of TSO, i.e., if two stores are ordered under the TSO semantics, they must be persisted in the same order. Thus, when a program is robust, one can reason about Px86 programs using TSO semantics, simplifying verification. However, there are efficient bug finding tools for checking robustness violations, as far as we are aware, there is currently no technique that enables robustness for Px86 to be checked for all possible executions.

Beillahi et al. [4] have notions of robustness for causally consistent transactions, which aims to reduce weak transactional consistency models to serialisability. This work studies transactional consistency as opposed to implementations of transactional memory, thus is orthogonal (but complementary) to our work.

Two recent works have developed modular proofs for durable opacity. Bila et al. [6] develop a durable library that supports transformation of *simulation-based proofs* of opaque TM implementations to *proofs* of durable opacity for the same TM that uses a persistency library. Given that TML has already been verified to be opaque [17], technically, such a proof could be reused. However, the library currently only supports the stronger PSC memory model. Raad et al. [61] present another modularisation technique that builds on the PMDK transactional library [36], which provides support for failure atomic transactions, but not concurrency. In particular, they show that PMDK transactions can be embedded within an STM to achieve both failure atomicity and thread safety, including under Px86, with validation performed using the FDR model checker [23]. Our intention is to directly support proofs of durable opacity, rather than rely on a third party.

9 Conclusions

In this work, we presented a revised version of the Px86 model [12] and PIEROGI [8], which allows reasoning about Px86 programs even after a system crash. Subsequently, we presented a durably opaque STM implementation under Px86 ($\text{dTML}_{\text{Px86}}$) and demonstrated a proof technique based on refinement for establishing correctness.

A possible extension of this work is exploring the connection between durable opacity and contextual refinement. This is particularly relevant in the case of persistent STM implementations like $\text{dTML}_{\text{Px86}}$, which are primarily used as libraries. Relevant work in the context of C11 has been conducted by Dalvandi and Dongol [16], demonstrating the insufficiency of TMS2 in providing client guarantees under the weak memory model of C11. In this work, a more adequate specification is proposed which constitutes an adaptation of TMS2, along with a program logic for verifying client programs. In the context of persistent memory, Khyzha and Lahav [45] have introduced a correctness criterion for contextual refinement.

We believe that the methodology can also be applied to construct a program logic for other weak persistent memory models such as the PArmv8 model. A potential starting point for this, could be the PArmv8 view-based semantics presented in Cho et al. [12].

Finally, formalising more weak correctness conditions for persistent STMs (e.g., buffered durable opacity, which can be defined in the same fashion as buffered durable linearisability [38]) as well as exploring their performance implications can be an interesting subject for future work.

A Overview of the thread-state views for Px86_{view}

Below, we provide a short description of the views of the thread state of Px86_{view}. We denote the pre-state state as σ, the post-state as σ', and the executing thread as t.

View: $\mathsf{coh} : \textsc{Loc} \rightarrow \mathbb{N}$

Moved by: **store**, **load**, **CAS**: Both a **store** and a successful **CAS** on x update $\sigma.\mathbb{T}(t).\mathsf{coh}(x)$ to match the length of the memory in the pre-state. A **load** and a failed **CAS** on x updates $\sigma.\mathbb{T}(t).\mathsf{coh}(x)$ to the timestamp of the message of the read value.

Purpose: In conjunction with $\mathsf{v_{rNew}}$, we use coh to determine the range of observable values by t for the specified location. When a memory message with the location x is about to be added to the memory by a **store** or a successful **CAS** operation, its timestamp in the post-state is equal to $\sigma.\mathbb{T}(t).\mathsf{coh}(x)$. Additionally, a message that is accessed by a **load** or **CAS** instruction must have a timestamp that is greater than or equal to the value of $\sigma.\mathbb{T}(t).\mathsf{coh}(x)$.

View: $\mathsf{v_{rNew}} : \mathbb{N}$

Moved by: **mfence**, **load**(external), **CAS**(fail-external/success): When t executes an **mfence**, $\sigma.\mathbb{T}(t).\mathsf{v_{rNew}}$ is updated to the timestamp of the latest write performed by t, provided that it is greater than the current value of $\mathsf{v_{rNew}}$ ($\sigma.\mathbb{T}(t).\mathsf{v_{rNew}}$). When t executes an external **load** or an external failed **CAS**, $\mathsf{v_{rNew}}$ is updated to the timestamp of the read message, again provided that it is greater than the current value of $\mathsf{v_{rNew}}$. When t executes a successful **CAS**, it updates $\sigma.\mathbb{T}(t).\mathsf{v_{rNew}}$ to the length that memory had in the pre-state.

Purpose Together with coh determines the set of visible values for the given location to t. No memory message that is read by t (via a **load** or a **CAS** instruction) obtains a timestamp that is overwritten from the ($\sigma.\mathbb{T}(t).\mathsf{v_{rNew}}$)'s perspective.

View: $\mathsf{v_{pReady}} : \mathbb{N}$

Moved by: **load**(external), **CAS**(fail-external/success), **mfence**, **sfence**: Instructions **load**(external), **CAS**(fail-external/success) and **mfence** update $\mathsf{v_{pReady}}$ in the same way that they update $\mathsf{v_{rNew}}$. The **sfence** instruction updates $\sigma.\mathbb{T}(t).\mathsf{v_{pReady}}$ in similar manner as **mfence**.

Purpose: It is used to order **load sfence**, **mfence** and **CAS** instructions with subsequent **flush**$_{\text{opt}}$ instructions.

View: $\mathsf{v_{pAsync}} : \textsc{Loc} \to \mathbb{N}$

Moved by: **flush**, **flush**$_{\text{opt}}$: When t executes a **flush** on x, $\sigma.\mathbb{T}(t).\mathsf{v_{pAsync}}(x)$ is updated to the timestamp of the latest write performed by t, provided that it is greater than $\sigma.\mathbb{T}(t).\mathsf{v_{pAsync}}(x)$. When t executes a **flush**$_{\text{opt}}$ on x, $\mathsf{v_{pAsync}}(x)$ is updated to the maximum between $\sigma.\mathbb{T}(t).\mathsf{coh}(x)$, $\sigma.\mathbb{T}(t).\mathsf{v_{pReady}}(x)$ and $\sigma.\mathbb{T}(t).\mathsf{v_{pAsync}}(x)$.

Purpose: Determines the set of values that may hold for a given location in persistent memory after the execution of an **sfence** preceded by the execution of a **flush**$_{\text{opt}}$. Any memory message the value of which is about to be persisted after the execution of a barrier (**sfence**, **mfence**, **CAS**), is not overwritten from the $\sigma'.\mathbb{T}(t).\mathsf{v_{pAsync}}$'s perspective in the post-state of a **flush**$_{\text{opt}}$ execution.

View: $\mathsf{v_{pCommit}} : \textsc{Loc} \to \mathbb{N}$

Moved by: **flush**, **CAS**(success), **mfence** and **sfence**: A **flush** on x updates $\sigma.\mathbb{T}(t).\mathsf{v_{pCommit}}(x)$ to the maximum between the timestamp of the latest write by t, and $\sigma.\mathbb{T}(t).\mathsf{v_{pCommit}}(x)$. Instructions **sfence** and **mfence** update $\sigma.\mathbb{T}(t).\mathsf{v_{pCommit}}(x)$ of all $x \in \textsc{Loc}$ to the maximum between $\sigma.\mathbb{T}(t).\mathsf{v_{pAsync}}(x)$ and $\sigma.\mathbb{T}(t).\mathsf{v_{pCommit}}(x)$. A successful **CAS** instruction updates $\sigma.\mathbb{T}(t).\mathsf{v_{pCommit}}(x)$ to the length that the memory had in the pre-state.

Purpose: Contributes to determining the set of values that may hold for a given location in persistent memory. The set of values for a location x that a thread can observe in persistent memory is common for all the threads. The set is determined by the maximum value of $\sigma.\mathbb{T}(t).\mathsf{v_{pCommit}}(x)$ among all the threads ($\bigsqcup \sigma.\mathsf{v_{pCommit}}(x)$). No memory message whose value reached the persistent memory after the execution of a persistent barrier or a **flush** instruction has a timestamp that is overwritten from the $\bigsqcup \sigma.\mathsf{v_{pCommit}}(x)$'s perspective after the completion of **flush** persist barrier execution.

B Hoare logic rules for atomic statements and stability of view-based assertions

In this section, we present a selection of the proof rules we employ to demonstrate the correctness of dTML$_{\text{Px86}}$. All the proof rules utilised in our refinement proof have been mechanised in Isabelle/HOL.

In Fig. 9, we provide a selection of rules for atomic statements where the statement is assumed to be exectuted by thread t. The first column presents the pre/post condition triple, the second column specifies any additional constraints, and the third column consists of labels that are used to reference these rules in our descriptions below. It should be noted that unless explicitly stated as a constraint, we do not assume that threads, locations, and values are distinct.

Rule $\mathsf{LP_1}$ states that if the thread view of t for x is the set S, then after the execution of a **load** instruction to x the value read belongs to the visible in the pre-state values of t for x

Precondition	Statement	Postcondition	Const.	Ref.
$\{[x]_t = S\}$	$r := \textbf{load}\ x$	$\{r \in S \wedge [x]_t \subseteq S\}$		LP_1
$\{[x]_t = \{u\}\}$		$\{r = \vec{x}\}$		LP_2
$\{true\}$	$\textbf{store}\ x\ v$	$\{[x]_t = \{v\}\}$		SP_1
$\{[x]_{t'} = S\}$		$\{[x]_{t'} = S \cup \{v\}\}$	$t \neq t'$	SP_2
$\{[x]^{\mathsf{A}}_{t'} = S\}$		$\{[x]^{\mathsf{A}}_{t'} = S \cup \{v\}\}$		SP_3
$\{[x]^{\mathsf{P}} = S\}$		$\{[x]^{\mathsf{P}} = S \cup \{v\}\}$		SP_4
$\{true\}$		$\{\mathsf{LE}(x) = \lvert M\rvert - 1 \wedge \vec{x} = v\}$		SP_5
$\{true\}$		$\left\{\begin{array}{l}\mathsf{LE}_{\mathsf{coh}}(x,t,x) = \lvert M\rvert - 1 \\ \wedge M[\mathsf{LE}_{\mathsf{coh}}(x,t,x)] \equiv \langle x := v\rangle\end{array}\right\}$		SP_6
$\{[x]_t = S \vee [x]^{\mathsf{A}}_t = S\}$	$\textbf{flush}_{\text{opt}}\ x$	$\{[x]^{\mathsf{A}}_t \subseteq S\}$		FP
$\{[x]^{\mathsf{A}}_t = S \vee [x]^{\mathsf{P}} = S\}$	$\textbf{sfence}$	$\{[x]^{\mathsf{P}} \subseteq S\}$		SFP
$\{[y]_{t'} = S\}$	$a := \textbf{CAS}\ x\ e_1\ e_2$	$\{[y]_{t'} \subseteq S\}$	$x \neq y$	CS_1
$\{true\}$		$\{a \Rightarrow \mathsf{LE}_{\mathsf{coh}}(x,t,x) = \lvert M\rvert - 1\}$		CS_2
$\{true\}$		$\left\{\begin{array}{l}a \Rightarrow \vec{x} = e_2 \wedge \\ M[\mathsf{LE}_{\mathsf{coh}}(x,t,x)] \equiv \langle x := e_2\rangle\end{array}\right\}$		CS_3
$\{true\}$		$\left\{\begin{array}{l}a \Rightarrow [y]_t = \{\vec{y}\} \wedge \\ M[\mathsf{LE}_{\mathsf{coh}}(x,t,y)] \equiv \langle y := \vec{y}\rangle\end{array}\right\}$		CS_4
$\{[x]^{\mathsf{A}}_t = S\}$		$\{a \Rightarrow [x]^{\mathsf{A}}_t = S \cup \{e_2\}\}$		CS_5
$\{[x]^{\mathsf{P}} = S\}$		$\{a \Rightarrow [x]^{\mathsf{P}} = S \cup \{e_2\}\}$		CS_6
$\{\vec{x} = v\}$		$\{\vec{x} = v \vee \vec{x} = e_2\}$		CS_7
$\{true\}$	$\textbf{Crash}$	$\{M[0](x) = \vec{x}\}$		C_1
$\{true\}$		$\{[x]^{\mathsf{P}}, [x]^{\mathsf{A}}_t, [x]_t = \{M[0](x)\}\}$		C_2
$\{[x]^{\mathsf{P}} = \{v\}\}$		$\{\vec{x} = v\}$		C_3

Fig. 9 Selected proof rules for atomic statements executed by thread t. Note t may be equal to t' and x may be equal to y unless explicitly ruled out.

Statement	Stable Assert.	Const.	Ref.
$r := \textbf{load}\ x$	$\{[y]_{t'} = S\}$	$t \neq t'$	LS_1
	$\{[y]^{\mathsf{P}} = S\}$		LS_2
	$\{[y]^{\mathsf{A}}_{t'} = S\}$		LS_3
	$\{r = k\}$		LS_4
	$\{\vec{y}\}$		LS_5
	$\{\mathsf{LE}_{\mathsf{coh}}(z, t', y)\}$	$t \neq t'$	LS_6
	$\{\mathsf{LE}(y)\}$		LS_7
$a := \textbf{CAS}\ x\ e_1\ e_2$	$\{[y]_{t'} = S\}$	$x \neq y \wedge t \neq t'$	FS_1
	$\{\vec{x} = v\}$	$\vec{x} \neq e_1$	FS_2
	$\{\mathsf{LE}_{\mathsf{coh}}(z, t', y)\}$	$t \neq t'$	FS_3
	$\{\mathsf{LE}(y)\}$	$x \neq y \vee \vec{x} \neq e_1$	FS_5
$\textbf{sfence}$	$\{[x]_{t'} = S\}$		SFS_1
	$\{\vec{x} = v\}$		SFS_2
	$\{\mathsf{LE}(x) = v\}$		SFS_3

Statement	Stable Assert.	Const.	Ref.
$\textbf{store}\ x\ v$	$\{[y]_{t'} = S\}$	$x \neq y$	WS_1
	$\{[y]^{\mathsf{P}} = S\}$	$x \neq y$	WS_2
	$\{[y]^{\mathsf{A}}_{t'} = S\}$	$x \neq y$	WS_3
	$\{r = k\}$		WS_4
	$\{\vec{y} = v\}$	$x \neq y$	WS_5
	$\{\mathsf{LE}_{\mathsf{coh}}(z, t', y) = v\}$	$x \neq z \vee t \neq t'$	WS_6
	$\{\mathsf{LE}(y) = v'\}$	$x \neq y$	WS_7
$\textbf{flush}_{\text{opt}}\ x$	$\{[y]_{t'} = S\}$		OS_1
	$\{[y]^{\mathsf{P}} = S\}$		OS_2
	$\{\vec{y} = v\}$		OS_3
	$\{\mathsf{LE}_{\mathsf{coh}}(z, t', y) = v\}$		OS_4
	$\{\mathsf{LE}(y) = v\}$		OS_5

Fig. 10 Selection of stable assertions for atomic statements executed by thread t. Note x may be equal to y and t may be equal to t' unless explicitly ruled out.

(S), and the thread view of t for x might become a subset of S (this is because of the possible update of the $\mathsf{coh}(x)$ and $\mathsf{v}_{\mathsf{rNew}}$ views). Rule LP_2 states that if the thread view of t for x contains only one element then after the execution of a **load** instruction to x the value read is surely the value of the last write at x. This is ensured by our well-formedness condition for Px86$_{view}$ which state that the thread, asynchronous and persistent view for a location x will always contain the last written value on x.

Rules SP_1-SP_6 refer to the **store** instruction. By SP_1, after t executes a **store** of value v to x, its only visible value for x becomes v. However, as stated in rule SP_2 any other thread continues to see the previously written values on x as well as the last written value v. Similarly, the *asynchronous view* for x of any thread (SP_3), and the *persistent view* for x (SP_4) are updated to contain the newly written value v. Rule SP_5 states that in the post-state the timestamp of the last message in memory with location x ($\mathsf{LE}(x)$), becomes the index of the last message in memory (|M|-1). In addition, the last written value at x ($\vec{x}$) becomes equal to v. Since the coherence view of x ($\mathsf{coh}(x)$) becomes equal to $|M|-1$ the expressions $\mathsf{LE}(x)$ and $\mathsf{LE}_{\mathsf{coh}}(x, t, x)$ are equivalent in the post-state. Therefore, as stated in rule SP_6, in the post-state $\mathsf{LE}_{\mathsf{coh}}(x, t, x)$ becomes equal to $|M| - 1$ and its value becomes equal to v.

Rule FP refers to the $\textbf{flush}_{\text{opt}}$ instruction and it states that the asynchronous view of x for t in the post-state is equal or a subset of its asynchronous view and its thread view in the pre-state.

By rule SFP, after the execution of an **sfence** instruction by t the persistent view of x becomes equal to or a subset of the asynchronous view of t for x and the *persistent view* of x in the pre-state.

Rules CS_1–CS_7 refer to the **CAS** instruction. A returned value true (resp. false) indicates a **CAS** success (resp. failure). Rule CS_1 states that given that $x \neq y$ the thread view of t for x after executing a **CAS** instruction is a subset or equal to its thread view for x in the pre-state. Rules CS_2–CS_6 describe the conditions that hold in case of a **CAS** success. In brief, when a **CAS** succeeds, it stores at x the value of e_2. Similar to the **store** instruction post conditions, in the post-state $\mathsf{LE}(x)$ becomes equivalent to the $\mathsf{LE}_{\mathsf{coh}}(x, t, x)$ expression and equal to $|M| - 1$ (rule CS_2). Moreover, its value is updated to e_2, which is the last written value on x (rule CS_3). Most importantly after a successful execution of **CAS**, the thread view of all the locations for t is updated to include only their last written value (rule CS_4). Furthermore, the asynchronous view of t for x and the persistent view of x are updated to include e_2 (rules CS_5, CS_6). Finally, rule CS_7 states that in the post-state the last written value at x either remains the same, indicating a **CAS** failure, or it changes to e_2.

Rules C_1–C_3 concern the **Crash** event. Rule C_1 states that in the post-crash state the initial message of the memory maps its location to its last stored value. This is trivial to show as after a crash only a single value (namely, the one that was persisted prior to the crash) remains observable for each location in the memory. By rule C_2 the thread, asynchronous and persistent view for each location x in the post-crash state contain only the value to which it is mapped in the initial message. Finally C_3 states that if the persistent view of any location x include only one value v in the pre-crash state, the last stored value on x ($\vec{x}$) after a crash takes place, will definitely be v.

Figure 10 contains a selection of assertions (middle column) that are proven stable against the corresponding atomic statements (left column) taking into account the constraints mentioned in the right. An assertion P is stable over a statement a executed by t iff $\{P\}a\{P\}$ holds. These proof rules are mostly used for establishing global correctness of the $\text{dTML}_{\text{Px86}}$ annotation.

B.1 Program annotations of $\text{dTML}_{\text{Px86}}$

Below, we provide a summary of the $\text{dTML}_{\text{Px86}}$ program annotations, apart from the program annotation of `TMRead` that is demonstrated in Sect. 6.4. As in Sect. 6.4, we colour the assertions of transactions that haven't performed a read or write yet in green colour, the assertions of read-only transactions in pink colour, and the assertions of writing transactions in blue

```
TMBegin
$PBp$ : $\{\neg\mathsf{hasRead}_t \wedge \neg\mathsf{hasWritten}_t \wedge \mathsf{writer} \neq t\}$
  $Bp$ : do $\mathsf{loc}_t := \mathbf{load}\ \mathsf{glb}$
$PB1$ : $\left\{\begin{pmatrix}\neg\mathsf{hasRead}_t \wedge \neg\mathsf{hasWritten}_t \wedge \mathsf{writer} \neq t \\ \wedge (even(\mathsf{loc}_t) \Rightarrow (\mathsf{loc}_t = \overrightarrow{\mathsf{glb}} \Rightarrow (\forall y.\ [y]_t = \{\vec{y}\})))\end{pmatrix}\right\}$
  $B1$ : until $even(\mathsf{loc}_t)$;
         return $ok$;   $\{\mathsf{ready}_t\}$
```

Fig. 11 `TMBegin` annotation.

colour. Assertions that refer to more than one category of transactions are not highlighted. The yellow highlighted assertions capture the effects of the preceding instruction.

The TMBegin annotation

We start with discussing the annotation of the `TMBegin` operation. In the initial state, all the registers are initialised to zero, therefore both the $\mathsf{hasWritten}_t$ and $\mathsf{hasRead}_t$ registers are set to zero (indicating false). The implication at *PB1* states that if the value read for glb is even, and is consistent with the last write of glb then t's thread view for every locations contains only its last value. The program annotations for `TMRead` and `TMWrite` guarantee that a subsequent read or write operation can only succeed if loc_t remains consistent with the last value of glb after the execution of Bp. $PB1$ is adequate to esptablish ready_t, in particular its first disjunct (Fig. 11).

The TMWrite annotation

Fig. 12 depicts the `TMWrite` annotation. The check performed at Wp determines whether a transaction t has previously executed a write operation. If the number of locations loc_t written by t is even, it means that t has not performed any writes yet. Consequently, $PW1$ asserts that $\mathsf{hasWritten}_t$ is $false$ and $\mathsf{writer} \neq t$. Additionally, $PW1$ ensures that if t is a writer, the asynchronous view for t of all locations in *log*, except for location x, is maximal. This guarantees that if t becomes a writing transaction and executes an sfence at $C1$, all of its writes will be persisted. Location x is excluded because, between the write operation at x ($W7$) and the asynchronous flush of the new write ($W8$), the asynchronous view of x contains both its old value and the newly written value ($\vec{x}$). Finally, $PW1$ states that the address to be written (x) is not equal to glb, which is necessary to establish Property 5.

Next, t attempts to acquire the single global versioned lock glb by executing **CAS** at $W1$. A successful **CAS** operation sets the $\mathsf{hasWritten}_t$ register to $true$, indicating that t has become a writing transaction. As stated in $PW2$, in this case, the last written value at glb is set to loc_t incremented by one, and the thread view of t for all memory locations is updated to include only their last written values. On the other hand, if **CAS** fails, it indicates the presence of another concurrent writing transaction, causing t to abort.

The subsequent execution of $W3$ increments loc_t by one. Therefore, according to $PW4$, loc_t becomes equal to the last value of glb. Additionally, the auxiliary variable writer is set to t.

Lines $W4$–$W9$ encompass the following operations: updating the *log* ($W4$–$W6$), performing the write at x ($W7$), and subsequently asynchronously flushing it ($W8$). The corresponding assertions remain unchanged, except for the final condition of $PW6$. This condition states that x is going to be updated to its last written value (c_t) in *log*. Establishing the condition

```
TMWrite$(x, v)$
$PWp$ : $\{\mathsf{ready}_t\}$
  $Wp$ : $\textbf{if}\ \ even(\mathsf{loc}_t)\ \textbf{then}$
$PW1$ : $\left\{\begin{array}{l} even(\mathsf{loc}_t) \wedge \neg\mathsf{hasWritten}_t \wedge \mathsf{writer} \neq t \\ \wedge(\mathsf{writer} = t \Rightarrow (\forall y \in \mathbf{dom}(log).\ x \neq y \Rightarrow\ [y]_t^{\mathsf{A}} = \{\vec{y}\})) \wedge x \neq \mathsf{glb} \end{array}\right\}$
  $W1$ :    $\mathsf{hasWritten}_t := \mathbf{CAS}\ \mathsf{glb}\ \mathsf{loc}_t\ (\mathsf{loc}_t + 1);$
$PW2$ :    $\left\{\begin{array}{l} (\mathsf{hasWritten}_t \Rightarrow (\forall y.\ [y]_t = \{\vec{y}\}) \wedge Succ(\mathsf{loc}_t) = \vec{\mathsf{glb}}) \\ \wedge(\mathsf{writer} = t \Rightarrow (\forall y \in \mathbf{dom}(log).\ x \neq y \Rightarrow\ [y]_t^{\mathsf{A}} = \{\vec{y}\})) \wedge x \neq \mathsf{glb} \end{array}\right\}$
  $W2$ :    $\textbf{if}\ \ \mathsf{hasWritten}_t\ \ \textbf{then}$
$PW3$ :    $\left\{\begin{array}{l} \mathsf{hasWritten}_t \wedge (\forall y.\ [y]_t = \{\vec{y}\}) \wedge Succ(\mathsf{loc}_t) = \vec{\mathsf{glb}} \\ \wedge(\mathsf{writer} = t \Rightarrow (\forall y \in \mathbf{dom}(log).\ x \neq y \Rightarrow\ [y]_t^{\mathsf{A}} = \{\vec{y}\})) \wedge x \neq \mathsf{glb} \end{array}\right\}$
  $W3$ :      $\langle \mathsf{loc}_t := \mathsf{loc}_t + 1, \mathsf{writer} := t \rangle$
            $\textbf{else return}\ aborted;\ \{\mathsf{true}\}$
$PW4$ : $\left\{\begin{array}{l} odd(\mathsf{loc}_t) \wedge \mathsf{loc}_t = \vec{\mathsf{glb}} \wedge (\forall y \in \mathbf{dom}(log).\ x \neq y \Rightarrow [y]_t^{\mathsf{A}} = \{\vec{y}\}) \wedge \mathsf{writer} = t \\ \wedge\mathsf{hasWritten}_t \wedge (\forall y.\ [y]_t = \{\vec{y}\}) \wedge x \neq \mathsf{glb} \end{array}\right\}$
  $W4$ : $\textbf{if}\ \neg log.\mathbf{contains}(x)\ \ \textbf{then}$
$PW5$ : $\left\{\begin{array}{l} odd(\mathsf{loc}_t) \wedge \mathsf{loc}_t = \vec{\mathsf{glb}} \wedge (\forall y \in \mathbf{dom}(log).\ x \neq y \Rightarrow [y]_t^{\mathsf{A}} = \{\vec{y}\}) \wedge \mathsf{writer} = t \\ \wedge\mathsf{hasWritten}_t \wedge (\forall y.\ [y]_t = \{\vec{y}\}) \wedge x \neq \mathsf{glb} \end{array}\right\}$
  $W5$ :    $c_t := \mathbf{load}\ x;$
$PW6$ :    $\left\{\begin{array}{l} odd(\mathsf{loc}_t) \wedge \mathsf{loc}_t = \vec{\mathsf{glb}} \wedge (\forall y \in \mathbf{dom}(log).\ x \neq y \Rightarrow [y]_t^{\mathsf{A}} = \{\vec{y}\}) \wedge \mathsf{writer} = t \\ \wedge\mathsf{hasWritten}_t \wedge (\forall y.\ [y]_t = \{\vec{y}\}) \wedge x \neq \mathsf{glb} \wedge c_t = \vec{x} \end{array}\right\}$
  $W6$ :    $log.\mathbf{update}(x, c_t);$
$PW7$ : $\left\{\begin{array}{l} odd(\mathsf{loc}_t) \wedge \mathsf{loc}_t = \vec{\mathsf{glb}} \wedge (\forall y \in \mathbf{dom}(log).\ x \neq y \Rightarrow [y]_t^{\mathsf{A}} = \{\vec{y}\}) \wedge \mathsf{writer} = t \\ \wedge\mathsf{hasWritten}_t \wedge (\forall y.\ [y]_t = \{\vec{y}\}) \end{array}\right\}$
  $W7$ : $\mathbf{store}\ x\ v;$
$PW8$ : $\left\{\begin{array}{l} odd(\mathsf{loc}_t) \wedge \mathsf{loc}_t = \vec{\mathsf{glb}} \wedge (\forall y \in \mathbf{dom}(log).\ x \neq y \Rightarrow [y]_t^{\mathsf{A}} = \{\vec{y}\}) \wedge \mathsf{writer} = t \\ \wedge\mathsf{hasWritten}_t \wedge (\forall y.\ [y]_t = \{\vec{y}\}) \end{array}\right\}$
  $W8$ : $\mathbf{flush}_{\mathrm{opt}}\ x;$
         $\textbf{return}\ ok;\ \ \{\mathsf{ready}_t\}$
```

Fig. 12 TMWrite annotation.

```
TMCommit
$PCp$ : $\{\mathsf{ready}_t\}$
  $Cp$ : $\textbf{if}\ \ odd(\mathsf{loc}_t)\ \textbf{then}$
$PC1$ : $\left\{\begin{array}{l} \mathsf{hasWritten}_t \wedge \mathsf{writer} = t \wedge (\forall y.\ [y]_t = \{\vec{y}\}) \wedge \mathsf{loc}_t = \vec{\mathsf{glb}} \\ \wedge(\forall y \in \mathbf{dom}(log).[y]_t^{\mathsf{A}} = \{\vec{y}\}) \end{array}\right\}$
  $C1$ : $\mathbf{sfence};$
$PC2$ : $\left\{\begin{array}{l} \mathsf{hasWritten}_t \wedge \mathsf{writer} = t \wedge (\forall y.\ [y]_t = \{\vec{y}\}) \wedge \mathsf{loc}_t = \vec{\mathsf{glb}} \\ \wedge(\forall y \in \mathbf{dom}(log).[y]^{\mathsf{P}} = \{\vec{y}\}) \end{array}\right\}$
  $C2$ :    $log.\mathbf{empty}();$
$PC3$ : $\left\{ \mathsf{hasWritten}_t \wedge \mathsf{writer} = t \wedge (\forall y.\ [y]_t = \{\vec{y}\}) \wedge \mathsf{loc}_t = \vec{\mathsf{glb}} \right\}$
  $C3$ :    $\langle \mathbf{store}\ \mathsf{glb}\ (\mathsf{loc}_t + 1),$
            $\mathsf{writer} := None \rangle$
  $Cr$ : $\textbf{return}\ commit;\ \ \{\mathsf{true}\}$
```

Fig. 13 TMCommit annotation.

```
TMRecover
$PRec1 : \left\{\begin{array}{l}\mathsf{writer} = \bot \wedge (\forall y.\ [y]_{syst} = \{\vec{y}\}) \\ \wedge(\forall ts \in \mathbf{dom}(M).ts > 0 \Rightarrow M[ts].\mathsf{loc} \neq \mathsf{glb})\end{array}\right\}$
 $Rec1 :$ $\mathbf{while}\ \neg log.\mathbf{isEmpty}()$
$PRec2 : \left\{\begin{array}{l}\mathbf{dom}(log) \neq \{\} \wedge \mathsf{writer} = \bot \wedge (\forall y.\ [y]_{syst} = \{\vec{y}\}) \\ \wedge(\forall ts \in \mathbf{dom}(M).ts > 0 \Rightarrow M[ts].\mathsf{loc} \neq \mathsf{glb})\end{array}\right\}$
 $Rec2 :$   $c_{syst} := log.\mathbf{getKey}();$
$PRec3 :$  $\left\{\begin{array}{l}c_{syst} \in \mathbf{dom}(log) \wedge \mathsf{writer} = \bot \wedge (\forall y.\ [y]_{syst} = \{\vec{y}\}) \\ \wedge\ (\forall ts \in \mathbf{dom}(M).ts > 0 \Rightarrow M[ts].\mathsf{loc} \neq \mathsf{glb})\end{array}\right\}$
 $Rec3 :$   $\mathbf{store}\ c_{syst}\ log.\mathbf{getVal}(c_{syst});$
$PRec4 :$  $\left\{\begin{array}{l}c_{syst} \in \mathbf{dom}(log) \wedge \mathsf{writer} = \bot \wedge (\forall y.\ [y]_{syst} = \{\vec{y}\}) \\ \wedge\ (\forall ts \in \mathbf{dom}(M).ts > 0 \Rightarrow M[ts].\mathsf{loc} \neq \mathsf{glb})\end{array}\right\}$
 $Rec4 :$   $\mathbf{flush}_{\mathrm{opt}}\ c_{syst};$
$PRec5 :$  $\left\{\begin{array}{l}c_{syst} \in \mathbf{dom}(log) \wedge \mathsf{writer} = \bot \wedge (\forall y.\ [y]_{syst} = \{\vec{y}\}) \\ \wedge [c_{syst}]^{\mathsf{A}}_{syst} = \{\vec{c_{syst}}\} \wedge\ (\forall ts \in \mathbf{dom}(M).ts > 0 \Rightarrow M[ts].\mathsf{loc} \neq \mathsf{glb})\end{array}\right\}$
 $Rec5 :$   $\mathbf{sfence};$
$PRec6 :$  $\left\{\begin{array}{l}c_{syst} \in \mathbf{dom}(log) \wedge \mathsf{writer} = \bot \wedge (\forall y.\ [y]_{syst} = \{\vec{y}\}) \\ \wedge [c_{syst}]^{\mathsf{P}} = \{\vec{c_{syst}}\} \wedge\ (\forall ts \in \mathbf{dom}(M).ts > 0 \Rightarrow M[ts].\mathsf{loc} \neq \mathsf{glb})\end{array}\right\}$
 $Rec6 :$   $log.\mathbf{update}(c_{syst}, \bot);$
$PRec7 : \left\{\begin{array}{l}\mathbf{dom}(log) = \{\} \wedge \mathsf{writer} = \bot \wedge (\forall y.\ [y]_{syst} = \{\vec{y}\}) \\ \wedge\ (\forall ts \in \mathbf{dom}(M).ts > 0 \Rightarrow M[ts].\mathsf{loc} \neq \mathsf{glb})\end{array}\right\}$
 $Rec7 : c_{syst} := \mathbf{load}\ \mathsf{glb};$
$PRec8 : \left\{\begin{array}{l}c_{syst} = M[0](\mathsf{glb}) \wedge \mathbf{dom}(log) = \{\} \wedge \mathsf{writer} = \bot \\ \wedge(\forall y.\ [y]_{syst} = \{\vec{y}\}) \wedge\ (\forall ts \in \mathbf{dom}(M).ts > 0 \Rightarrow M[ts].\mathsf{loc} \neq \mathsf{glb})\end{array}\right\}$
 $Rec8 : \mathbf{if}\ \ even(c_{syst})\ \mathbf{then}$
$PRec9 : \left\{\begin{array}{l}even(c_{syst}) \wedge \mathbf{dom}(log) = \{\} \wedge \mathsf{writer} = \bot \wedge c_{syst} = M[0](\mathsf{glb}) \\ \wedge(\forall y.\ [y]_{syst} = \{\vec{y}\}) \wedge\ (\forall ts \in \mathbf{dom}(M).ts > 0 \Rightarrow M[ts].\mathsf{loc} \neq \mathsf{glb})\end{array}\right\}$
 $Rec9 :$   $\langle \mathbf{store}\ \mathsf{glb}\ c_{syst} + 2,$
            $\mathsf{recGlb} := c_{syst} + 2\rangle\{pc_{syst} = Rec_{complete}\}$
$PRec9 : \left\{\begin{array}{l}odd(c_{syst}) \wedge \mathbf{dom}(log) = \{\} \wedge \mathsf{writer} = \bot \wedge c_{syst} = M[0](\mathsf{glb}) \\ \wedge(\forall y.\ [y]_{syst} = \{\vec{y}\}) \wedge\ (\forall ts \in \mathbf{dom}(M).ts > 0 \Rightarrow M[ts].\mathsf{loc} \neq \mathsf{glb})\end{array}\right\}$
$Rec10 : \mathbf{else}\ \langle \mathbf{store}\ \mathsf{glb}\ (c_{syst} + 1),$
              $\mathsf{recGlb} := c_{syst} + 1\rangle\{pc_{syst} = Rec_{complete}\}$
```

Fig. 14 `TMRecover` annotation.

ready, particularly its third disjunct, from $PW8$ is straightforward. It should be noted that after the execution of $W8$, the asynchronous view of x contains only its last written value (as per the FP rule in Fig. 9). The combination of the above rule with $PW8$ is sufficient to establish that the asynchronous view of t for all locations in the domain of log contains only their last written value ($(\forall y \in \mathbf{dom}(log).\ x \neq y \implies [y]^{\mathsf{A}}_t = \vec{y})$).

The TMCommit annotation

Fig. 13 illustrates the `TMCommit` annotation. Transactions that have not performed any read or write and read-only transactions commit without any further check. In the case of a writing transaction t, according to the assertion ready, loc_t is odd, $\mathsf{hasWritten}_t$ is true, $\mathsf{writer} = t$, the thread view of t for any location y includes only the last stored value at y, and the asynchronous view of t for any location y that belongs to the domain of log includes only

Table 1 TML history events where $t \in$ TID, $x \in$ LOC and $v \in$ VAL

Invocations	Possible matching responses
inv_t(`TMBegin`)	res_t(`TMBegin`(ok)), res_t(`TMBegin`($abort$))
inv_t(`TMCommit`)	res_t(`TMCommit`($commit$)), res_t(`TMCommit`($abort$))
inv_t(`TMRead`(x))	res_t(`TMRead`(v)), res_t(`TMRead`($abort$))
inv_t(`TMWrite`(x, v))	res_t(`TMWrite`(ok)), res_t(`TMWrite`($abort$))

the last stored value at y. It is worth noting that the locations that belong to the domain of log, are the only locations that have been updated by a writing transaction t. As seen at the postcondition of $C1$, after the execution of **sfence** the asynchronous views of t for the aforementioned locations become equal to their persistent views. Having the above stated prior to emptying the log (i.e. at $PC2$) is sufficient for establishing locally Property 6. Property 6, guarantees that during the execution and commit of read-only transactions, and after writing transactions commit, the value can be observed in persistent memory for any location x apart from glb is deterministic and equal to the last written value on x.

The TMRecover annotation

Fig. 14 illustrates the `TMRecover` annotation. The `TMRecover` annotation serves three purposes. **(1)** It provides sufficient information about the memory state after a crash event and during the `TMRecover` process, in order to establish that Property 1 and Property 2 locally hold. The above is enabled by the assertion: $\forall ts \in \mathbf{dom}(M).\ ts > 0 \implies M[ts].\mathsf{loc} \neq \mathsf{glb}$, which ensures that during recovery all the memory messages apart for the initial one, represent writes to locations different from glb. For showing this during copying and emptying the log we use the Property 5. **(2)** It guarantees the consistency of memory upon completion of the recovery process. This is accomplished by Property 6 in combination with rules C1 and C3 (see Figure 9). By applying Property 6 and the aforementioned rules, any location y within the initial message is mapped to its persisted value $\vec{y}$, which represents the last value written to y by a committed transaction prior to the system crash. Moreover, the recovery process sequentially restores all the locations recorded in the log. The `TMRecover` annotation guarantees that the recovered values correspond to those stored in the log. **(3)** It guarantees that by the completion of the recovery process the last written value in glb is even and greater than its initial value.

C Durable opacity

We now provide a series of definitions that gradually lead to the formal definition of *durable opacity* [5], which is the correctness criterion against which we validate our STM implementation (dTML$_{\text{Px86}}$).

Histories

Correctness conditions for concurrent objects, such as TM, are predominantly defined over *histories* of an implementation, which is a sequence of events (invocations and responses) that records all the interactions between the object and its clients. Typical TM operations are summarised in Fig. 1. For a response event e, we let $rval(e) \in \{ok, \bot, abort, commit\}$ denote the value returned by e.

In an NVM setting, a history must also record system-wide crash events, *crash*. Thus, a history H, in this case, has the form $H = h_0 c_0 h_1 c_1 \dots h_{n-1} c_{n-1} h_n c_n$, where each h_i is a history (containing no crash events) and c_i is the ith crash event. We refer to each h_i as an *era* of H.

We use standard list notation for histories. For a history, h, $h_{|t}$ is the projection onto the events of the transaction t and $h[i..j]$ is the subsequence of h from $h(i)$ to $h(j)$ (inclusive). We let $ops(h)$ denote the subsequence of h with all crashes removed.

A history h is *alternating* if $h = \epsilon$ or is an alternating sequence of invocation and matching response events starting with an invocation.

In a history h, the *real-time order* of transactions t_1 and t_2 is defined as $t_1 \prec_h t_2$ if t_1 is a completed transaction and the last event of t_1 in h occurs before the first event of t_2 in h. If neither $t_1 \prec_h t_2$ nor $t_2 \prec_h t_1$ holds, we consider transactions t_1 and t_2 to be concurrent. A history h is *non-interleaved* if it does not contain concurrent transactions.

Well-formed Histories

A crash-free history is *well-formed* iff for every $t \in \textsc{Tid}$, either $h_{|t} = \varepsilon$, or $h_{|t} = \langle s_0, \dots, s_m \rangle$ is an alternating history such that $s_0 = inv_t(\texttt{TMBegin})$, for all $0 < i \leq m$, event $s_i \neq inv_t(\texttt{TMBegin})$ and $rval(s_i) \notin \{commit, abort\}$. A history h is *durably well-formed* iff $ops(h)$ is well-formed and every transaction identifier appears in at most one era.

Sequential specification

We now describe the sequential semantics of TM implementations, which we note is by definition crash-free.

Let $h = ev_0, \dots, ev_{2n-1}$ be a crash-free sequence of alternating invocation and matching response events starting with an invocation and ending with a response. We say h is *valid* iff there exists a sequence of stores $\sigma_0, \dots, \sigma_n \in (\textsc{Loc} \to \textsc{Val})^*$ such that $\sigma_0(x) = 0$ for all $x \in \textsc{Loc}$, and for all i such that $0 \leq i < n$ and $t \in \textsc{Tid}$:

(1) If $ev_{2i} = inv_t(\texttt{TMWrite}(x, v))$ and $ev_{2i+1} = res_t(\texttt{TMWrite}(ok))$ then $\sigma_{i+1} = \sigma_i[x := v]$,
(2) If $ev_{2i} = inv_t(\texttt{TMRead}(x))$ and $ev_{2i+1} = res_t(\texttt{TMRead}(v))$ then $\sigma_i(x) = v$ and $\sigma_{i+1} = \sigma_i$,
(3) For all other pairs of events (reads and writes with an abort response, as well as begin and commit events) we require $\sigma_{i+1} = \sigma_i$.

Let hs be a crash-free non-interleaved history and i an index of hs. Let hs' be the projection of $hs[0..(i-1)]$ onto all events of committed transactions plus the events of the transaction to which $hs(i)$ belongs. Then we say hs is *legal* at i whenever hs' is valid. We say *hs is legal* iff it is legal at each index i. A well-formed history hs is *sequential* if it is non-interleaved and legal. We denote by $\mathcal{S}$ the set of all possible well-formed sequential histories.

Durable opacity

A concurrent history is a sequence of events corresponding to operations executed by different transactions. A history of transactions may be incomplete, i.e., it may contain pending operations, represented by invocations that do not have matching responses, or it may obtain transactions that have not yet invoked a commit operation (*live* transactions). To enable reasoning about these, we use a function $complete(h)$ that constructs the set of all possible completions of h by appending successful matching responses ($res_t(\texttt{TMCommit}(commit))$) for some pending `TMCommit` invocations, appending matching abort responses to all other pending operations and appending an `TMCommit` invocation and aborted response event to all live transactions.

Informally, a history h that contains crash events is *durably opaque* if the resulting history after removing the crash events ($ops(h)$) is *opaque* (as defined by Guerraoui and Kapalka [33]) and each thread id appearing in h appears to only one of its crash free eras. The latter condition corresponds to the *well-formedness* requirement of *durable opacity*.

Definition 3 A crash-free history h is *end-to-end opaque* iff for some $hc \in complete(h)$, there exists a sequential history $hs \in \mathcal{S}$ such that for all $t \in \textsc{Tid}$, $h_{|t} = hs_{|t}$ and $\prec_{hc} \subseteq \prec_{hs}$. A history h is *opaque* iff each prefix h' of h is end-to-end opaque.

Definition 4 (Durable opacity) A history h is *durably opaque* iff it is durably well-formed and $ops(h)$ is opaque. A TM implementation is *opaque* iff each of its histories is opaque.

Acknowledgements Vafeiadi Bila is supported by VeTSS. Dongol is supported by VeTSS and EPSRC grants EP/Y036425/1, EP/X037142/1, EP/X015149/1, EP/V038915/1, and EP/R025134/2.

Author Contributions Vafeiadi Bila did all the proofs in consultation with Dongol. Both authors wrote and revised the main manuscript text.

Declarations

Conflict of interest The authors declare no Conflict of interest.

References

1. Abdulla PA, Atig MF, Bouajjani A et al (2021) Deciding reachability under persistent x86-TSO. Proc ACM Program Lang 5(POPL):1–32. https://doi.org/10.1145/3434337
2. Armstrong A, Dongol B, Doherty S (2017) Proving opacity via linearizability: a sound and complete method. In: Bouajjani A, Silva A (eds) FORTE, LNCS, vol 10321. Springer, pp 50–66
3. Attiya H, Gotsman A, Hans S, et al (2013) A programming language perspective on transactional memory consistency. In: Fatourou P, Taubenfeld G (eds) PODC '13. ACM, pp 309–318. https://doi.org/10.1145/2484239.2484267
4. Beillahi SM, Bouajjani A, Enea C (2021) Robustness against transactional causal consistency. Log Methods Comput Sci 17(1). URL https://lmcs.episciences.org/7149
5. Bila E, Doherty S, Dongol B, et al (2020) Defining and verifying durable opacity: Correctness for persistent software transactional memory. In: Gotsman A, Sokolova A (eds) FORTE, LNCS, vol 12136. Springer, pp 39–58. https://doi.org/10.1007/978-3-030-50086-3_3

6. Bila E, Derrick J, Doherty S et al (2022) Modularising verification of durable opacity. Log Methods Comput Sci. https://doi.org/10.46298/lmcs-18(3:7)2022 https://doi.org/10.46298/lmcs-18(3:7)2022 https://doi.org/10.46298/lmcs-18(3:7)2022
7. Bila EV, Dongol B (2024) Isabelle/HOL files for "A verified durable transactional mutex lock for persistent x86-TSO". URL https://figshare.com/articles/thesis/DTML_correctness_proof/25037312
8. Bila EV, Dongol B, Lahav O et al (2022) View-based Owicki–Gries reasoning for persistent x86-tso. In: Sergey I (ed) ESOP, LNCS, vol 13240. Springer, pp 234–261. https://doi.org/10.1007/978-3-030-99336-8_9
9. Böhme S, Nipkow T (2010) Sledgehammer: judgement day. In: Giesl J, Hähnle R (eds) Automated reasoning, 5th international joint conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings, LNCS, vol 6173. Springer, pp 107–121. https://doi.org/10.1007/978-3-642-14203-1_9
10. Chajed T, Tassarotti J, Kaashoek MF, et al (2019) Verifying concurrent, crash-safe systems with perennial. In: Proceedings of the 27th ACM symposium on operating systems principles, pp 243–258
11. Chakrabarti DR, Boehm H, Bhandari K (2014) Atlas: leveraging locks for non-volatile memory consistency. In: Black AP, Millstein TD (eds) OOPSLA. ACM, pp 433–452. https://doi.org/10.1145/2660193.2660224
12. Cho K, Lee SH, Raad A, et al (2021) Revamping hardware persistency models: view-based and axiomatic persistency models for Intel-x86 and Armv8. In: Freund SN, Yahav E (eds) PLDI. ACM, pp 16–31. https://doi.org/10.1145/3453483.3454027
13. Coburn J, Caulfield AM, Akel A, et al (2011) Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In: Gupta R, Mowry TC (eds) ASPLOS. ACM, pp 105–118. https://doi.org/10.1145/1950365.1950380
14. Correia A, Felber P, Ramalhete P (2018) Romulus: efficient algorithms for persistent transactional memory. In: SPAA, pp 271–282
15. Dalessandro L, Dice D, Scott M, et al (2010) Transactional mutex locks. In: Euro-Par, Springer, pp 2–13
16. Dalvandi S, Dongol B (2022) Implementing and verifying release-acquire transactional memory in C11. Proc ACM Program Lang 6(OOPSLA2):1817–1844. https://doi.org/10.1145/3563352
17. Derrick J, Doherty S, Dongol B et al (2018) Mechanized proofs of opacity: a comparison of two techniques. Formal Asp Comput 30(5):597–625. https://doi.org/10.1007/s00165-017-0433-3
18. Derrick J, Doherty S, Dongol B et al (2021) Verifying correctness of persistent concurrent data structures: a sound and complete method. Formal Aspects Comput 33(4–5):547–573. https://doi.org/10.1007/s00165-021-00541-8
19. Dice D, Shalev O, Shavit N (2006) Transactional locking II. In: International symposium on distributed computing, Springer, pp 194–208
20. Doherty S, Groves L, Luchangco V et al (2013) Towards formally specifying and verifying transactional memory. Form Aspect Comput 25:769–799
21. Doherty S, Dongol B, Derrick J, et al (2016) Proving opacity of a pessimistic STM. In: Fatourou P, Jiménez E, Pedone F (eds) OPODIS, LIPIcs, vol 70. Schloss Dagstuhl–Leibniz–Zentrum für Informatik, pp 35:1–35:17
22. Dongol B, Derrick J (2015) Verifying linearisability: a comparative survey. ACM Comput Surv 48(2):19:1-19:43. https://doi.org/10.1145/2796550
23. Dongol B, Le-Papin J (2021) Checking opacity and durable opacity with FDR. In: Calinescu R, Pasareanu CS (eds) SEFM, LNCS, vol 13085. Springer, pp 222–242. https://doi.org/10.1007/978-3-030-92124-8_13
24. D'Osualdo E, Raad A, Vafeiadis V (2023) The path to durable linearizability. Proc ACM Program Lang 7(POPL):748–774. https://doi.org/10.1145/3571219
25. Dziuma D, Fatourou P, Kanellou E (2014) Consistency for transactional memory computing. Bull EATCS 113
26. Feijen WHJ, van Gasteren AJM (1999) On a method of multiprogramming. Monographs in computer science. Springer. https://doi.org/10.1007/978-1-4757-3126-2
27. Felber P, Gramoli V, Guerraoui R (2009) Elastic transactions. In: DISC, Springer, pp 93–107
28. Friedman M, Ben-David N, Wei Y, et al (2020) Nvtraverse: in NVRAM data structures, the destination is more important than the journey. In: Donaldson AF, Torlak E (eds) PLDI. ACM, pp 377–392. https://doi.org/10.1145/3385412.3386031
29. Friedman M, Petrank E, Ramalhete P (2021) Mirror: making lock-free data structures persistent. In: Freund SN, Yahav E (eds) PLDI. ACM, pp 1218–1232. https://doi.org/10.1145/3453483.3454105
30. Giles E, Doshi KA, Varman PJ (2015) Softwrap: a lightweight framework for transactional support of storage class memory. In: IEEE MSST. IEEE Computer Society, pp 1–14. https://doi.org/10.1109/MSST.2015.7208276

31. Gorjiara H, Luo W, Lee A, et al (2022) Checking robustness to weak persistency models. In: Jhala R, Dillig I (eds) PLDI. ACM, pp 490–505. https://doi.org/10.1145/3519939.3523723,
32. Gu J, Yu Q, Wang X, et al (2019) Pisces: a scalable and efficient persistent transactional memory. In: Malkhi D, Tsafrir D (eds) USENIX ATC. USENIX Association, pp 913–928
33. Guerraoui R, Kapalka M (2010) Principles of transactional memory. synthesis lectures on distributed computing theory, Morgan & Claypool Publishers. https://doi.org/10.2200/S00253ED1V01Y201009DCT004
34. Herlihy M, Wing JM (1990) Linearizability: a correctness condition for concurrent objects. ACM Trans Program Lang Syst 12(3):463–492. https://doi.org/10.1145/78969.78972
35. Imbs D, Raynal M (2012) Virtual world consistency: a condition for STM systems (with a versatile protocol with invisible read operations). Theor Comput Sci 444:113–127. https://doi.org/10.1016/j.tcs.2012.04.037
36. Intel (2022) Persistent memory development kit, libpmemobj library. URL https://pmem.io/pmdk/libpmemobj/
37. Intel Corporation (2021) Intel 64 and IA-32 architectures optimization reference manual
38. Izraelevitz J, Mendes H, Scott ML (2016) Linearizability of persistent memory objects under a full-system-crash failure model. In: Gavoille C, Ilcinkas D (eds) DISC, LNCS, vol 9888. Springer, pp 313–327. https://doi.org/10.1007/978-3-662-53426-7_23
39. Jeong J, Hong J, Maeng S, et al (2020) Unbounded hardware transactional memory for a hybrid dram/nvm memory system. In: MICRO, IEEE, pp 525–538
40. Joshi A, Nagarajan V, Viglas S, et al (2017) ATOM: atomic durability in non-volatile memory through hardware logging. In: HPCA. IEEE computer society, pp 361–372. https://doi.org/10.1109/HPCA.2017.50
41. Joshi A, Nagarajan V, Cintra M, et al (2018) Dhtm: durable hardware transactional memory. In: ISCA, IEEE, pp 452–465
42. Jung R, Krebbers R, Jourdan JH et al (2018) Iris from the ground up: a modular foundation for higher-order concurrent separation logic. J Funct Program 28:e20
43. Kammüller F, Wenzel M, Paulson LC (1999) Locales a sectioning concept for isabelle. In: TPHOLs, Springer, pp 149–165
44. Khyzha A, Lahav O (2021) Taming x86-TSO persistency. Proc ACM Program Lang 5(POPL):1–29. https://doi.org/10.1145/3434328
45. Khyzha A, Lahav O (2022) Abstraction for crash-resilient objects. In: Sergey I (ed) ESOP, LNCS, vol 13240. Springer, pp 262–289. https://doi.org/10.1007/978-3-030-99336-8_10
46. Kolli A, Pelley S, Saidi AG, et al (2016) High-performance transactions for persistent memories. In: Conte T, Zhou Y (eds) ASPLOS. ACM, pp 399–411. https://doi.org/10.1145/2872362.2872381
47. Krishnan RM, Kim J, Mathew A, et al (2020) Durable transactional memory can scale with timestone. In: ASPLOS, pp 335–349
48. Lamport L (1979) How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans Comput 28(9):690–691. https://doi.org/10.1109/TC.1979.1675439
49. Lesani M, Palsberg J (2014) Decomposing opacity. In: Kuhn F (ed) DISC, LNCS, vol 8784. Springer, pp 391–405. https://doi.org/10.1007/978-3-662-45174-8_27
50. Lesani M, Luchangco V, Moir M (2012) Putting opacity in its place. In: Workshop on the theory of transactional memory, pp 137–151
51. Liu M, Zhang M, Chen K, et al (2017) Dudetm: Building durable transactions with decoupling for persistent memory. In: Chen Y, Temam O, Carter J (eds) ASPLOS. ACM, pp 329–343.https://doi.org/10.1145/3037697.3037714
52. Lynch NA (1996) Distributed algorithms. Morgan Kaufmann
53. Matichuk D, Murray T, Wenzel M (2016) Eisbach: a proof method language for Isabelle. J Autom Reason 56(3):261–282. https://doi.org/10.1007/s10817-015-9360-2
54. Owicki SS, Gries D (1976) An axiomatic proof technique for parallel programs I. Acta Inform 6:319–340. https://doi.org/10.1007/BF00268134
55. Papadimitriou CH (1979) The serializability of concurrent database updates. J ACM (JACM) 26(4):631–653
56. Raad A, Doko M, Rozic L et al (2019) 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. https://doi.org/10.1145/3290381
57. Raad A, Wickerson J, Vafeiadis V (2019) Weak persistency semantics from the ground up: formalising the persistency semantics of armv8 and transactional models. Proc ACM Program Lang 3(OOPSLA):135:1-135:27. https://doi.org/10.1145/3360561

58. Raad A, Lahav O, Vafeiadis V (2020) Persistent Owicki–Gries reasoning: a program logic for reasoning about persistent programs on Intel-x86. Proc ACM Program Lang 4(OOPSLA):151:1-151:28. https://doi.org/10.1145/3428219
59. Raad A, Wickerson J, Neiger G et al (2020) Persistency semantics of the Intel-x86 architecture. Proc ACM Program Lang 4(POPL):11:1-11:31. https://doi.org/10.1145/3371079
60. Raad A, Maranget L, Vafeiadis V (2022) Extending Intel-x86 consistency and persistency: formalising the semantics of Intel-x86 memory types and non-temporal stores. Proc ACM Program Lang 6(POPL):1–31. https://doi.org/10.1145/3498683
61. Raad A, Lahav O, Wickerson J et al (2024) Intel PMDK transactions: Specification, validation and concurrency. In: Weirich S (ed) ESOP, LNCS, vol 14577. Springer, pp 150–179. https://doi.org/10.1007/978-3-031-57267-8_6
62. Ramalhete P, Correia A, Felber P, et al (2019) Onefile: a wait-free persistent transactional memory. In: DSN. IEEE, pp 151–163. https://doi.org/10.1109/DSN.2019.00028
63. Ren J, Zhao J, Khan SM, et al (2015) ThyNVM: enabling software-transparent crash consistency in persistent memory systems. In: Prvulovic M (ed) MICRO. ACM, pp 672–685. https://doi.org/10.1145/2830772.2830802
64. Sewell P, Sarkar S, Owens S et al (2010) x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. Commun ACM 53(7):89–97
65. Siek K, Wojciechowski PT (2022) Last-use opacity: a strong safety property for transactional memory with prerelease support. Distribut Comput 35(3):265–301. https://doi.org/10.1007/s00446-022-00420-2
66. Singh AK, Lahav O (2023) An operational approach to library abstraction under relaxed memory concurrency. Proc ACM Program Lang 7(POPL):1542–1572. https://doi.org/10.1145/3571246
67. Sun L, Lu Y, Shu J (2015) Dp^2: reducing transaction overhead with differential and dual persistency in persistent memory. In: Napoli CD, Salapura V, Franke H, et al (eds) CF. ACM, pp 24:1–24:8. https://doi.org/10.1145/2742854.2742864
68. Vindum SF, Birkedal L (2022) Spirea: a mechanized concurrent separation logic for weak persistent memory
69. Volos H, Tack AJ, Swift MM (2011) Mnemosyne: lightweight persistent memory. In: Gupta R, Mowry TC (eds) ASPLOS. ACM, pp 91–104. https://doi.org/10.1145/1950365.1950379
70. Wei Y, Ben-David N, Friedman M, et al (2022) Flit: a library for simple and efficient persistent algorithms. In: Lee J, Agrawal K, Spear MF (eds) PPoPP. ACM, pp 309–321. https://doi.org/10.1145/3503221.3508436