Defining and Verifying Durable Opacity: Correctness for Persistent Software Transactional Memory

Eleni Bila, Simon Doherty, Brijesh Dongol, John Derrick, Gerhard Schellhorn, Heike Wehrheim

Defining and Verifying Durable Opacity: Correctness for Persistent Software Transactional Memory ⋆

Eleni Bila[1], Simon Doherty[2], Brijesh Dongol[1], John Derrick[2], Gerhard Schellhorn[3], and Heike Wehrheim[4]

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

Abstract. Non-volatile memory (NVM), aka persistent memory, is a new paradigm for memory that preserves its contents even after power loss. The expected ubiquity of NVM has stimulated interest in the design of novel concepts ensuring correctness of concurrent programming abstractions in the face of persistency. So far, this has lead to the design of a number of persistent concurrent data structures, built to satisfy an associated notion of correctness: durable linearizability.
In this paper, we transfer the principle of durable concurrent correctness to the area of software transactional memory (STM). Software transactional memory algorithms allow for concurrent access to shared state. Like linearizability for concurrent data structures, opacity is the established notion of correctness for STMs. First, we provide a novel definition of durable opacity extending opacity to handle crashes and recovery in the context of NVM. Second, we develop a durably opaque version of an existing STM algorithm, namely the Transactional Mutex Lock (TML). Third, we design a proof technique for durable opacity based on refinement between TML and an operational characterisation of durable opacity by adapting the TMS2 specification. Finally, we apply this proof technique to show that the durable version of TML is indeed durably opaque. The correctness proof is mechanized within Isabelle.

1 Introduction

Recent technological advances indicate that future architectures will employ some form of *non-volatile memory* (NVM) that retains its contents after a system crash (e.g., power outage). NVM is intended to be used as an intermediate layer between traditional *volatile memory* (VM) and secondary storage, and has the potential to vastly improve system speed and stability. Software that uses NVM has the potential to be more robust; in case of a crash, a system state before the crash may be recovered using contents from NVM, as opposed to being restarted from secondary storage. However, because the same data is stored in both a volatile and non-volatile manner, and because NVM is updated at a slower rate than VM, recovery to a consistent state may not always

⋆ Bila and Dongol are supported by VeTSS project "Persistent Safety and Security". Dongol is supported by EPSRC grants EP/R019045/2 and EP/R032556/1. Derrick and Doherty are supported by EPSRC grant EP/R032351/1. Wehrheim is supported by DFG grant WE2290/12-1.

be possible. This is particularly true for concurrent systems, where coping with NVM requires introduction of additional synchronisation instructions into a program.

This observation has led to the design of the first *persistent* concurrent programming abstractions, so far mainly concurrent data structures. Together with these, the associated notion of correctness, i.e., linearizability [21], has been transferred to NVM. This resulted in the novel concept of *durable linearizability* [22]. A first proof technique for showing durable linearizability of concurrent data structures has been proposed by Derrick et al. [11].

Besides concurrent data structures, software transactional memory (STM) is the most important synchronization mechanism supporting concurrent access to shared state. STMs provide an *illusion of atomicity* in concurrent programs. The analogy of STM is with database transactions, which perform a series of accesses/updates to shared data (via read and write operations) atomically in an all-or-nothing manner. Similarly with an STM, if a transaction commits, all its operations succeed, and in the aborting case, all its operations fail. The now (mainly) agreed upon correctness criterion for STMs is *opacity* [20]. Opacity requires all transactions (including aborting ones) to agree on a single sequential history of committed transactions and the outcome of transactions has to coincide with this history.

In this paper, we transfer STM and opacity to the novel field of non-volatile memory. This entails a number of steps. First, the correctness criterion of opacity has to be adapted to cope with crashes in system executions. Second, STM algorithms have to be extended to deal with the coexistence of volatile and non-volatile memory during execution and need to be equipped with recovery operations. Third, proof techniques for opacity need to be re-investigated to make them usable for durable opacity. In this paper, we provide contributions to all three steps.

- For the first step, we define *durable opacity* out of opacity in the same way that durable linearizability has been defined based on linearizability. Durable opacity requires the executions of STMs to be opaque even if they are interspersed with crashes. This guarantees that the shared state remains consistent.
- We exemplify the second step by extending the Transactional Mutex Lock (TML) of Dalessandro et al. [8] to durable TML (DTML). To this end, TML needs to be equipped with a recovery operation and special statements to guarantee consistency in case of crashes. We do so by extending TML with a logging mechanism which allows to flush written, but volatile values to NVM during recovery.
- For the third step, we build on a proof technique for opacity based on refinement between IO-automata. This technique uses the automaton TMS2 [15] which has been shown to implement opacity [26] using the PVS interactive theorem prover. This automaton gives us a formal specification, which can be used as the abstract level in a proof of refinement. Furthermore, the IO-automaton framework is part of the standard Isabelle distribution. For use as a proof technique for durable opacity, TMS2 is extended with a crash operation (mimicing system crashes and their effect on memory) to yield DTMS2. The automaton DTMS2 is then proven to only have durably opaque executions. Thereby we obtain an operational characterisation of durable opacity.

invocations	possible matching responses
$inv_t(\texttt{TMBegin})$	$res_t(\texttt{TMBegin}(\texttt{ok}))$
$inv_t(\texttt{TMCommit})$	$res_t(\texttt{TMCommit}(\texttt{ok}))$, $res_t(\texttt{TMCommit}(\texttt{abort}))$
$inv_t(\texttt{TMRd}(x))$	$res_t(\texttt{TMRd}(v))$, $res_t(\texttt{TMRd}(\texttt{abort}))$
$inv_t(\texttt{TMWr}(x,v))$	$res_t(\texttt{TMWr}(\texttt{ok}))$, $res_t(\texttt{TMWr}(\texttt{abort}))$

Table 1. Events appearing in the histories of TML, where $t \in T$ is a transaction identifier, $x \in L$ is a location, and $v \in V$ a value

Finally, we bring all three steps together and apply our proof technique to show that durable TML is indeed durably opaque. This proof has been mechanized in the interactive prover Isabelle [32]. Our mechanized proof proceeds by encoding DTMS2 and DTML as IO-automata within Isabelle, then proving the existence of a forward simulation, which in turn has been shown to ensure trace refinement of IO-automata [28], and hence guarantees durable opacity of DTML.

2 Transactional Memory and Opacity

Software Transactional Memory (STM) provides programmers with an easy-to-use synchronisation mechanism for concurrent access to shared data, whereby blocks of code may be treated as transactions that execute with an illusion of atomicity. STMs usually provide a number of operations to programmers: operations to start (`TMBegin`) and commit a transaction (`TMCommit`), and operations to read and write shared data (`TMRd`, `TMWr`). These operations can be called (invoked) from within a client program (possibly with some arguments, e.g., the variable to be read) and then will return with a response. Except for operations that start transactions, all other operations might potentially respond with `abort`, thereby aborting the whole transaction.

A widely accepted correctness condition for STMs that encapsulates transactional phenomena is *opacity* [19,20], which requires all transactions, including aborting transactions to agree on a single sequential global ordering of transactions. Moreover, no transactional read returns a value that is inconsistent with the global ordering.

2.1 Histories

As standard in the literature, opacity is defined on the *histories* of an implementation. Histories are sequences of *events* that record all interactions between the implementation and its clients. An event is either an invocation (*inv*) or a response (*res*) of a transactional operation. For the TML implementation, possible invocation and *matching* response events are given in Table 1, where we assume T is a set of transaction identifiers, L a set of addresses (or locations) mapped to values from a set V.

The type $Mem \mathrel{\widehat{=}} L \to V$ describes the possible states of the shared memory. We assume that initially all addresses hold the value $0 \in V$.

We use the following notation on histories: for a history h, $h \restriction t$ is the projection onto the events of transaction t only and $h[i..j]$ the subsequence of h from $h(i)$ to $h(j)$

inclusive. For a response event e, we let $rval(e)$ denote the value returned by e; for instance $rval(\texttt{TMBegin}(\texttt{ok})) = \texttt{ok}$. If e is not a response event, then we let $rval(e) = \bot$.

We are interested in three different types of histories [2]. At the concrete level the TML implementation produces histories where the events are interleaved. At the abstract level we are interested in *sequential histories*, which are ones where there is no interleaving at any level - transactions are atomic: a transaction completes before the next transaction starts. As part of the proof of opacity we use an intermediate specification which has *alternating histories*, in which transactions may be interleaved but operations (e.g., reads, writes) are not interleaved.

A history h is *alternating* if $h = \epsilon$ or is an alternating sequence of invocation and matching response events starting with an invocation. For the rest of this paper, we assume each process invokes at most one operation at a time, and hence, assume that $h \restriction t$ is alternating for any history h and transaction t. Note that this does not necessarily mean h is alternating itself. Opacity is defined for well-formed histories, which formalises the allowable interaction between an STM implementation and its clients. For every t, $h \restriction t = \langle s_0, \ldots, s_m \rangle$ of a well-formed history is an alternating history such that $s_0 = inv_t(\texttt{TMBegin})$, for all $0 < i < m$, event $s_i \neq inv_t(\texttt{TMBegin})$ and $rval(s_i) \notin \{\texttt{commit}, \texttt{abort}\}$. Note that by definition, well-formedness disallows transaction identifiers from being reused. We say t is *committed* if $rval(s_m) = \texttt{commit}$ and *aborted* if $rval(s_m) = \texttt{abort}$. In these cases, the transaction t is *completed*, otherwise t is *live*. A history is *well-formed* if it consists of transactions only and there is at most one live transaction per process.

2.2 Opacity

Opacity [19,20] compares concurrent histories generated by an STM implementation to sequential histories and can be seen as a strengthening of serializability to accommodate aborted transactions. Below, we first formalise the sequential history semantics, then consider opaque histories.

Sequential history semantics. A sequential history has to ensure that the behaviour is meaningful with respect to the reads and writes of the transactions.

Definition 1 (Valid history). *Let $h = ev_0, \ldots, ev_{2n-1}$ be a sequence of alternating invocation and matching response events starting with an invocation and ending with a response.*

We say h is valid *if there exists a sequence of states $\sigma_0, \ldots, \sigma_n$ such that $\sigma_0(l) = 0$ for all $l \in L$, and for all i such that $0 \leq i < n$ and $t \in T$:*

1. *if $ev_{2i} = inv_t(\texttt{TMWr}(l, v))$ and $ev_{2i+1} = res_t(\texttt{TMWr}(\texttt{ok}))$ then $\sigma_{i+1} = \sigma_i[l := v]$,*
2. *if $ev_{2i} = inv_t(\texttt{TMRd}(l))$ and $ev_{2i+1} = res_t(\texttt{TMRd}(v))$ then $\sigma_i(l) = v$ 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 $\sigma_{i+1} = \sigma_i$.*

We write $[\![h]\!](\sigma)$ if σ is a sequence of states that makes h valid (since the sequence is unique, if it exists, it can be viewed as the semantics of h).

The point of TM is that the effect of the writes only takes place if the transaction commits. Writes in a transaction that abort do not affect the memory. However, all reads, including those executed by aborted transactions, must be consistent with previously committed writes. Therefore, only some histories of an object reflect ones that could be produced by a TM. We call these the *legal* histories, and they are defined as follows.

Definition 2 (Legal histories). *Let hs be a 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.*

This allows us to define sequentiality for a single history, which we lift to the level of specifications.

Definition 3 (Sequential history). *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.*

Opaque histories. Opacity is defined by matching a concurrent history to a sequential history such that (a) both histories consist of the same events, and (b) the real-time order of transactions is preserved. For (b), the *real-time order* on transactions t_1 and t_2 in a history h 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.

A given concrete history may be incomplete, i.e., it may contain pending operations, represented by invocations that do not have matching responses. Some of these pending operations may have taken effect, and others may not. The corresponding sequential history however must decide: either by adding a suitable matching response event for the pending invocation (the effect has taken place), or by removing the pending invocation (no effect yet). Therefore, we define a function *complete*(h) that constructs all possible completions of h by appending matching responses for some pending invocations and removing all the other pending invocations. This is similar to the treatment of completions in the formalisation of linearizability [21]. The sequential history then must have the same events as those of one of the results returned by *complete*(h).

Definition 4 (Opaque history). *A 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* $t \in T$*,* $hc\lceil t = hs\lceil t$ *and* $\prec_{hc} \subseteq \prec_{hs}$*. A history h is* opaque *iff each prefix h′ of h is end-to-end opaque; a set of histories* $\mathcal{H}$ *is* opaque *iff each* $h \in \mathcal{H}$ *is opaque; and an STM implementation is* opaque *iff its set of histories is opaque.*

3 STMs over Persistent Memory

We now consider STMs over a non-volatile memory model comprising two layers: a *volatile store*, whose contents are wiped clean when a system crashes (e.g., due to power loss), and a *persistent store*, whose state is preserved after a crash and available for use upon reboot. During normal program execution, contents of the volatile store may be

transferred to the persistent store by the system. The main idea behind programs for this memory model is to include a *recovery procedure* that executes over the persistent store and resets the system into a consistent (safe) state. To achieve this, a programmer can control transfer of information from volatile to persistent store using a `FLUSH(a)` operation, ensuring that the information in address `a` is saved in the persistent store.

For STMs, we introduce a new notion of consistency: *durable opacity* which we define in Section 3.1. Durable opacity extends opacity [19, 20] in exactly the same way that durable linearizability [22] extends linearizability [21], namely a history that contains crashes is durably opaque precisely when the same history with crashes removed is opaque. We present an example STM implementation that satisfies durable opacity in Section 3.2, extending Dalessandro et al.'s Transactional Mutex Lock [9].

3.1 Durable Opacity

Durable opacity is a correctness condition that is defined over *histories* that record the *invocation* and *response* events of operations executed on the transactional memory like opacity. Unlike opacity, durably opaque histories record system crash events, thus may take the form: $H = h_0 c_1 h_1 c_2 ... h_{n-1} c_n h_n$, where each h_i is a history (containing no crash events) and c_i is the ith crash event. Following Izraelevitz et al. [22], for a history h, we let $ops(h)$ denote h restricted to non-crash events, thus for H above, $ops(H) = h_0 h_1 \ldots h_{n-1} h_n$, which contains no crash events. We call the subhistory h_i the *i-th era* of h.

The definition of a well-formed history is now updated to include crash events. A history is *durably well-formed* iff $ops(h)$ is well formed and every transaction identifier appears in at most one era. Thus, we assume that when a crash occurs, all running transactions are aborted.

Definition 5 (Durably opaque history). *A history h is* durably opaque *iff it is durably well-formed and $ops(h)$ is opaque.*

3.2 Example: Durable Transactional Mutex Lock

We now develop a durably opaque STM: a persistent memory version of the Transactional Mutex Lock (TML) [8], as given in Fig. 1. TML adopts a strict policy for transactional synchronisation: as soon as one transaction has successfully written to a variable, all other transactions running concurrently will be aborted when they invoke another read or write operation. To enforce this policy, TML uses a global counter `glb` (initially 0) and local variable `loc`, which is used to store a copy of `glb`. Variable `glb` records whether there is a *live writing transaction*, i.e., a transaction that has started, has not yet ended nor aborted, and has executed (or is executing) a write operation. More precisely, `glb` is odd if there is a live writing transaction, and even otherwise. Initially, we have no live writing transaction and thus `glb` is 0 (and hence even).

A second distinguishing feature of TML is that it performs writes in an *eager* manner, i.e., it updates shared memory during the write operation[5]. This is potentially problematic in a persistent memory context since writes that have completed may not be

[5] This is in contrast to lazy implementations that defer transactional writes until the commit operation is executed (e.g., [9, 13]).

committed if a crash occurs prior to executing the commit operation. That is, writes of uncommitted transactions should not be seen by any transactions that start after a crash occurs. Our implementation makes use of an *undo log* mapping addresses to their persistent memory values prior to executing the first write operation for that address. Logged values are made persistent before the address is overwritten. Thus, if a crash occurs prior to a transaction committing, it is possible to recover the transaction to a safe state by undoing uncommitted transactional writes.

Operation `TMBegin` copies the value of `glb` into its local variable `loc` and checks whether `glb` is even. If so, the transaction is started, and otherwise, the process attempts to start again by rereading `glb`. A `TMRead` operation succeeds as long as `glb` equals `loc` (meaning no writes have occurred since the transaction began), otherwise it aborts the current transaction. The first execution of `TMWrite` attempts to increment `glb` using a `cas` (compare-and-swap), which atomically compares the first and second parameters, and sets the first parameter to the third if the comparison succeeds. If the `cas` attempt fails, a write by another transaction must have occured, and hence, the current transaction aborts. Otherwise `loc` is incremented (making its value odd) and the write is performed. Note that because `loc` becomes odd after the first successful write, all successive writes that are part of the same transaction will perform the write directly after testing `loc` at line $W1$. Further note that if the `cas` succeeds, `glb` becomes odd, which prevents other transactions from starting, and causes all concurrent live transactions still wanting to read or write to abort. Thus a writing transaction that successfully updates `glb` effectively locks shared memory. Operation `TMEnd` checks to see if a write has occurred by testing whether `loc` is odd. If the test succeeds, `glb` is set to `loc+1`. At line `E2`, `loc` is guaranteed to be equal to `glb`, and therefore this update has the effect of incrementing `glb` to an even value, allowing other transactions to begin.

Our implementation uses a durably linearizable [11, 22] set or map data structure `log`, such as the one described by Zuriel et al. [38]. A durably linearizable operation is guaranteed to take effect in persistent memory prior to the operation returning. In Fig. 1, we use use operations `pinsert()`, `pempty()` and `pdelete()` to stress that these operations are durably linearizable.

Our durable TML algorithm (DTML) makes the following adaptations to TML. Note the the operations build on a model of a crash that resets volatile memory to persistent memory.

- Within a write operation writing to address `addr`, prior to modifying the value at `addr`, we record the existing address-value pair in `log`, provided that `addr` does not already appear in the undo log (lines `W4` and `W5`). After updating the value (which updates the value of `addr` in the volatile store), the update is flushed to persistent memory prior to the write operation returning (line `W7`).
- We introduce a recovery operation that checks for a non-empty log and transfers the logged values to persistent memory, undoing any writes that have completed (but not committed) before the crash occurred. Since a crash could occur during recovery, we transfer values from the undo log to persistent memory one at a time.
- In the commit operation, we note that we distinguish a committing transaction as one with an odd value for `loc`. For a writing transaction, the log must be cleared by setting it to the empty log (line `E2`). Note that this is the point at which a writ-

```
TMBegin:
B1 do loc := glb
B2 until even(loc);
   return ok;

TMRead(addr):
R1 val := *addr;
R2 if (glb = loc) then
     return val;
   else return abort;

Recover():
C1 while ¬ log.isEmpty()
C2   SOME (addr, val).
        (addr, val) ∈ log;
C3   *addr := val ;
C4   FLUSH(addr) ;
C5   log.pdelete((addr, val));
C6 glb := 0
```

```
TMCommit:
E1 if odd(loc) then
E2   log.pempty();
E3   glb := loc + 1;
   return commit;

TMWrite(addr,val):
W1 if even(loc) then
W2   if ¬ cas(glb,loc,loc+1) then
        return abort;
W3   else loc++;
W4 if ∀ v. (addr, v) ∉ log then
W5    log.pinsert((addr, *addr));
W6 *addr := val;
W7 FLUSH(addr);
   return ok;
```

Fig. 1. A durable Transactional Mutex Lock (DTML). Initially: `glb = 0, log = emptyLog()`. Line numbers for `return` statements are omitted and we use `*addr` for the value of `addr`

ing transaction has definitely committed since any subsequent crash and recovery would no longer undo the writes of this transaction.

4 Proving Durable Opacity

Previous works [1, 2, 14, 17] have considered proofs of opacity using the operational TMS2 specification [15], which has been shown to guarantee opacity [26]. The proofs show refinement of the implementation against the TMS2 specification using either forward or backward simulation. For durable opacity, we use a similar proof strategy. In Section 4.3, we develop the DTMS2 operational specification, a durable version of the TMS2 specification, that we prove satisfies durable opacity. Then, in Section 5, we establish a simulation between DTML and DTMS2.

4.1 Background: IOA, Refinement and Simulation

We use Input/Output Automata (IOA) [29] to model both the implementation, DTML, and the specification, DTMS2.

Definition 6. *An* Input/Output Automaton (IOA) *is a labeled transition system A with a set of* states $states(A)$*, a set of* actions $acts(A)$*, a set of* start states $start(A) \subseteq states(A)$*, and a* transition relation $trans(A) \subseteq states(A) \times acts(A) \times states(A)$ *(so that the actions label the transitions).*

The set $acts(A)$ is partitioned into input actions $input(A)$, output actions $output(A)$ and internal actions $internal(A)$. The internal actions represent events of the system that are not visible to the external environment. The input and output actions are externally visible, representing the automaton's interactions with its environment. Thus, we define the set of *external actions*, $external(A) = input(A) \cup output(A)$. We write $s \xrightarrow{a}_A s'$ iff $(s, a, s') \in trans(A)$.

An *execution* of an IOA A is a sequence $\sigma = s_0 a_0 s_1 a_1 s_2 \ldots s_n a_n s_{n+1}$ of alternating states and actions, such that $s_0 \in start(A)$ and for all states s_i, $s_i \xrightarrow{a_i}_A s_{i+1}$. A *reachable* state of A is a state appearing in an execution of A. An *invariant* of A is any superset of the reachable states of A (equivalently, any predicate satisfied by all reachable states of A). A *trace* of A is any sequence of (external) actions obtained by projecting the external actions of any execution of A. The set of traces of A, denoted $traces(A)$, represents A's externally visible behaviour.

For automata C and A, we say that C is a *refinement* of A iff $traces(C) \subseteq traces(A)$. We show that C is a refinement of A by proving the existence of a *forward simulation*, which enables one to check step correspondence between the transitions of C and those of A. The definition of forward simulation we use is adapted from that of Lynch and Vaandrager [28].

Definition 7. *A* forward simulation *from a concrete IOA C to an abstract IOA A is a relation $R \subseteq states(C) \times states(A)$ such that each of the following holds.*

Initialisation. $\forall\, cs \in start(C).\ \exists as \in start(A).\ R(cs, as)$

External step correspondence.

$$\forall\, cs \in reach(C), as \in reach(A), a \in external(C), cs' \in states(C).$$
$$R(cs, as) \wedge cs \xrightarrow{a}_C cs' \Rightarrow \exists as' \in states(A).\ R(cs', as') \wedge as \xrightarrow{a}_A as'$$

Internal step correspondence.

$$\forall\, cs \in reach(C), as \in reach(A), a \in internal(C), cs' \in states(C).$$
$$R(cs, as) \wedge cs \xrightarrow{a}_C cs' \Rightarrow$$
$$R(cs', as) \vee \exists a' \in internal(A), as' \in states(A).\ R(cs', as') \wedge as \xrightarrow{a'}_A as'$$

Forward simulation is *sound* in the sense that if there is a forward simulation between A and C, then C refines A [28, 30].

4.2 IOA for DTML

We now provide the IOA model of DTML. The state of DTML (Fig. 1) comprises global (shared) variables $glb \in \mathbb{N}$ (modelling `glb` in volatile memory); $log \in L \rightharpoonup V$, where $\rightharpoonup$ denotes a partial function (modelling `log` in persistent memory); the volatile memory store $vstore \in L \to V$; and the persistent memory store $pstore \in L \to V$. We also use the following transaction-local variables: the program counter $pc \in T \to PC$, $loc \in T \to \mathbb{N}$, the input address $addr \in T \to V$, the input value $val \in T \to V$. We also make use of an auxiliary variable *writer* whose value is either the transaction id of the current writing transaction (if one exists), or *None* (if no writing transaction is currently running).

Execution of the program is modelled by defining an IOA transition for each atomic step of Fig. 1, using the values of pc_t (for transaction t) to model control flow. Each

action that starts a new operation or returns from a completed operation is an external action. The crash action is also external. All other actions (including flush and recovery) are internal actions.

To model system behaviours (crash, system flush and recovery), we reserve a special transaction id *syst*. A crash and system flush is always enabled, and hence can always be selected for execution. Recovery steps are enabled after a crash has taken place and are only executed by *syst*. The effect of a flush is to copy the value of the address being flushed from *vstore* to *pstore*. Note that a flush can also be executed at specific program locations. In DTML, a flush of `addr` occurs at lines `W7` and `C5`. The effect of a crash is to perform the following:

- set the volatile store to the persistent store (since the volatile store is lost),
- set the program counters of all *in-flight transactions* (i.e., transactions that have started but not yet completed) to *aborted* to ensure that these transaction identifiers are not reused after the system is rebooted, and
- set the status of *syst* to `C1` to model that a recovery is now in progress.

In our model, it is possible for a system to crash during recovery. However, no new transaction may start until after the recovery process has completed.

4.3 IOA for DTMS2

In this section, we describe the DTMS2 specification, an operational model that ensures durable opacity, which is based on TMS2 [15]. TMS2 itself has been shown to strictly imply opacity [26], and hence has been widely used as an intermediate specification in the verification of transactional memory implementations [1, 2, 12, 14].

We let $f \oplus g$ denote functional override of f by g, e.g., $f \oplus \{x \mapsto u, y \mapsto v\} = \lambda k.$ **if** $k = x$ **then** u **elseif** $k = y$ **then** v **else** $f(k)$.

Formally, DTMS2 is specified by the IOA in Fig. 2, which describes the required ordering constraints, memory semantics and prefix properties. We assume a set L of locations and a set V of values. Thus, a memory is modelled by a function of type $L \to V$. A key feature of DTMS2 (like TMS2) is that it keeps track of a *sequence* of memory states, one for each committed writing transaction. This makes it simpler to determine whether reads are consistent with previously committed write operations. Each committing transaction containing at least one write adds a new memory version to the end of the memory sequence. However, unlike TMS2, following [11], the memory state is considered to be the persistent memory state. Interestingly, the volatile memory state need not be modelled.

The state space of DTMS2 has several components. The first, *mems* is the sequence of *memory* states. For each transaction t there is a program counter variable pc_t, which ranges over a set of *program counter values*, which are used to ensure that each transaction is well-formed, and to ensure that each transactional operation takes effect between its invocation and response. There is also a a *begin index* variable $beginIdx_t$, that is set to the index of the most recent memory version when the transaction begins. This variable is critical to ensuring the real-time ordering property between transactions. Finally, there is a *read set*, $rdSet_t$, and a *write set*, $wrSet_t$, which record the values that the transaction has read and written during its execution, respectively.

State variables:
$mems$: $seq(L \to V)$, initially satisfying $\mathrm{dom}\, mems = \{0\}$ and $initMem(mems(0))$
pc_t : $PCVal$, for each $t \in T$, initially $pc_t =$ notStarted for all $t \in T$
$beginIdx_t$: $\mathbb{N}$ for each $t \in T$, unconstrained initially
$rdSet_t$: $L \nrightarrow V$, initially empty for all $t \in T$
$wrSet_t$: $L \nrightarrow V$, initially empty for all $t \in T$

Transition relation:

$inv_t(\texttt{TMBegin})$
Pre: $pc_t =$ notStarted
Eff: $pc_t :=$ beginPending
$beginIdx_t := len(mems) - 1$

$inv_t(\texttt{TMRd}(l))$
Pre: $pc_t =$ ready
Eff: $pc_t :=$ doRead(l)

$inv_t(\texttt{TMWr}(l, v))$
Pre: $pc_t =$ ready
Eff: $pc_t :=$ doWrite(l, v)

$inv_t(\texttt{TMCommit})$
Pre: $pc_t =$ ready
Eff: $pc_t :=$ doCommit

$resp_t(\texttt{abort})$
Pre: $pc_t \notin \{$notStarted, ready, commitResp, committed, aborted$\}$
Eff: $pc_t :=$ aborted

$\texttt{DoCommitReadOnly}_t(n)$
Pre: $pc_t =$ doCommit
$\mathrm{dom}(wrSet_t) = \varnothing$
$validIdx(t, n)$
Eff: $pc_t :=$ commitResp

$\texttt{DoRead}_t(l, n)$
Pre: $pc_t =$ doRead(l)
$l \in \mathrm{dom}(wrSet_t) \vee validIdx(t, n)$
Eff: if $l \in \mathrm{dom}(wrSet_t)$ then
$pc_t :=$ readResp$(wrSet_t(l))$
else $v := mems(n)(l)$
$pc_t :=$ readResp(v)
$rdSet_t := rdSet_t \oplus \{l \to v\}$

$resp_t(\texttt{TMBegin})$
Pre: $pc_t =$ beginPending
Eff: $pc_t :=$ ready

$resp_t(\texttt{TMRd}(v))$
Pre: $pc_t =$ readResp(v)
Eff: $pc_t :=$ ready

$resp_t(\texttt{TMWr})$
Pre: $pc_t =$ writeResp
Eff: $pc_t :=$ ready

$resp_t(\texttt{TMCommit})$
Pre: $pc_t =$ commitResp
Eff: $pc_t :=$ committed

$\texttt{DoWrite}_t(l, v)$
Pre: $pc_t =$ doWrite(l, v)
Eff: $pc_t :=$ writeResp
$wrSet_t := wrSet_t \oplus \{l \to v\}$

$\texttt{DoCommitWriter}_t$
Pre: $pc_t =$ doCommit
$rdSet_t \subseteq last(mems)$
Eff: $pc_t :=$ commitResp
$mems := mems \frown (last(mems) \oplus wrSet_t)$

$crash_t$
Pre: $t = syst$
Eff: $pc := \lambda\, t : T.$
if $t \neq syst \wedge$
$pc_t \notin \{notStarted, committed\}$
then $aborted$
$mems = \langle last(mems) \rangle$

where $validIdx(t, n) \;\widehat{=}\; beginIdx_t \leq n < len(mems) \wedge rdSet_t \subseteq mems(n)$

Fig. 2. The state space and transition relation of DTMS2, which extends TMS2 with a crash event

The read set is used to determine whether the values that have been read by the transaction are consistent with the same version of memory (using *validIdx*). The write set, on the other hand, is required because writes in DTMS2 are modelled using *deferred update* semantics: writes are recorded in the transaction's write set, but are not published to any shared state until the transaction commits.

The *crash* action models both a crash and a recovery. We require that it is executed by the system thread *syst*. It sets the program counter of every in-flight transaction to *aborted*, which prevents these transactions from performing any further actions in the era following the crash (for the generated history). Note that since transaction identifiers are not reused, the program counters of completed transactions need not be set to any special value (e.g., *crashed*) as with durable linearizability [11]. Moreover, after restarting, it must not be possible for any new transaction to interact with memory states prior to the crash. We therefore reset the memory sequence to be a singleton sequence containing the last memory state prior to the crash.

The following theorem ensures that DTMS2 can be used as an intermediate specification in our proof method. We provide a proof sketch below. The full proof may be found in the appendix of [5].

Theorem 1. *Each trace of* DTMS2 *is durably opaque.*

Proof (Sketch). First we recall that TMS2 is exactly the same as the automaton in Fig. 2, but without a crash operation. The proof proceeds by showing that for any history $h \in traces(\text{DTMS2})$, we have that $ops(h) \in traces(\text{TMS2})$. Then since $ops(h)$ is opaque, we have that h is durably opaque. We establish a formal relationship between h and $ops(h)$ by establishing a *weak simulation* between DTMS2 and TMS2 such that $\{ops(h) \mid h \in traces(\text{DTMS2})\} \subseteq traces(\text{TMS2})$. The simulation is weak since the external *crash* action in DTMS2 has no matching counterpart in TMS2.

The simulation relation we use captures the following. Any transaction t of DTMS2 that is aborted due to a crash will set pc_t to *aborted* without executing $resp_t(\texttt{abort})$. This difference can easily be compensated by the simulation relation. A second difference is that *mems* is reset to $last(mems)$ in DTMS2 when a crash occurs, and hence there is a mismatch between *mems* in DTMS2 and in TMS2. Let *ds* be a state of DTMS2 and *as* a state of TMS2. To compensate for the difference between *ds.mems* and *as.mems*, we introduce an auxiliary variable "*allMems*" to *ds* that records memories corresponding to all committed writing transactions in DTMS2. We have the property that *ds.mems* of DTMS2 is a suffix of *ds.allMems* and that $ds.allMems = as.mems$.

5 Durable Opacity of DTML

We now describe the simulation relation used in the Isabelle proof.[6]

Our simulation relation is divided into two relations: a *global relation globalRel*, and a transactional relation *txnRel*. The global relation describes how the shared states of the two automata are related, and the transaction relation specifies the relationship

[6] All Isabelle theory files related to this proof may be downloaded from [5].

between the state of each transaction in the concrete automaton, and that of the transaction in the abstract automaton. The simulation relation itself is then given by:

$$\mathit{simRel}(cs, as) = \mathit{globalRel}(cs, as) \wedge \forall\, t \in T \bullet \mathit{txnRel}(cs, as, t)$$

We first describe *globalRel*, which assumes the following auxiliary definitions where *cs* is the concrete state (of DTML) and *as* is the abstract state (of DTMS2). These definitions are used to compensate for the fact that the commit of a writing transaction in the DTML algorithm takes effect (i.e., linearizes) at line `E2` when the log is set to empty.

$$\begin{aligned}
\mathit{writes}(cs, as) &= \textbf{if } cs.\mathit{writer} = t \wedge pc_t \neq E3 \textbf{ then } as.\mathit{wrSet}_t \textbf{ else } \varnothing \\
\mathit{logical_glb}(cs) &= \textbf{if } cs.\mathit{writer} = t \wedge pc_t = E3 \textbf{ then } cs.\mathit{glb} + 1 \textbf{ else } cs.\mathit{glb} \\
\mathit{write_count}(cs) &= \left\lfloor \frac{\mathit{logical_glb}(cs)}{2} \right\rfloor
\end{aligned}$$

Function $\mathit{writes}(cs, as)$ returns the (abstract) write set of the writing transaction. This is the write set of the writing transaction, *t*, in the abstract state *as* provided *t* hasn't already linearized its commit operation, and is the empty set otherwise. Function $\mathit{logical\_glb}(cs)$ compensates for a lagging value of *glb* after a writing transaction's commit operation is linearized. Namely, it returns the *glb* incremented by 1 if a writer is already at $E3$. Finally, $\mathit{write\_count}(cs)$ is used to determine the number of committed writing transactions in *cs* since the most recent crash since $cs.\mathit{glb}$ is initially 0 and reset to 0 by the recovery operation, and moreover, $cs.\mathit{glb}$ is incremented twice by each writing transaction: once at line `W2` and again at line `E2` when the writing transaction commits.

Relation *globalRel* comprises three main parts. We assume a program counter value *RecIdle* which is true for pc_{syst} iff *syst* is not executing the recovery procedure.

$$\begin{aligned}
&\mathit{globalRel}(cs, as) = \\
&(pc_{\mathit{syst}} = \mathit{RecIdle} \Rightarrow cs.\mathit{vstore} = (\mathit{last}(as.\mathit{mems}) \oplus \mathit{writes}(cs, as)) \wedge && (1)\\
&\qquad \mathit{write_count}(cs) + 1 = \mathit{length}(as.\mathit{mems}))) \wedge && (2)\\
&(cs.\mathit{vstore} \oplus cs.\mathit{log}) = \mathit{last}(\mathit{mems}(as)) \wedge && (3)\\
&\forall\, t.t \neq \mathit{syst} \wedge cs.pc_t = \mathit{NotStarted} \Rightarrow as.pc_t = \mathit{NotStarted} && (4)
\end{aligned}$$

Conditions (1) and (2) assume that a recovery procedure is not in progress. By (1), the concrete volatile store is the last memory in *as.mems* overwritten with the write set of an in-flight writing transaction that has not linearized its commit operation. By (2), the number of memories recorded in the abstract state (since the last crash) is equal to $\mathit{write\_count}(cs) + 1$. By (3), the last abstract (persistent) store can by calculated from *cs.vstore* by overriding it with the mappings in log. Note that this is equivalent to undoing all uncommitted transactional writes. Finally, (4) ensures that every identifier for a transaction that has not started at the concrete level also has not started at the abstract level.

We turn now to *txnRel*. Its specification is very similar to the specification of *txnRel* in the proof of TML [10]. Therefore, we only provide a brief sketch below; an interested reader may consult [10] for further details. Part of *txnRel* maps concrete program counters to their abstract counterparts, which enables steps of the concrete program to be

matched with abstract steps. For example, concrete *pc* values `W1`, `W2`, ..., `W6` correspond to abstract *pc* value doWrite($cs.addr_t, cs.val_t$), whereas `W7` corresponds to writeResp, indicating that, in our proof, execution of line `W6` corresponds to the execution of an abstract $\texttt{DoWrite}_t(cs.addr_t, cs.val_t)$ operation. Moreover, as in the proof of TML [10], a set of assertions are introduced to establish $as.validIdx(t, write\_count(cs))$ for all in-flight transactions t, which ensures that each transactional read and write is valid with respect to some memory snapshot.

Relation *txnRel* must also provide enough information to enable linearization of a commit operation against the correct abstract step. Note that DTMS2 distinguishes between read-only and writing transactions by checking emptiness of the write set of the committing transaction. To handle this, we exploit the fact that in DTML, writing transactions have an odd *loc* value if the `cas` at line `W2` is successful and `loc` is incremented at `W3`, indicating that a writing transaction is in progress.

Finally, *txnRel* must ensure that the recovery operation is such that the volatile store matches the last abstract store in *mems* prior to the crash. To achieve this, we require that $length(as.mems) = 1$ when *syst* is executing the recovery procedure, and the volatile store for the address being flushed at `C3` matches the abstract state before the crash, i.e., $cs.vstore(cs.addr_t) = ((as.mems)(0))(cs.addr_t)$. Since the recovery loop only terminates after the log is emptied, this ensures that the concrete memory state is consistent with the abstract memory prior to executing any transactions after a crash has occurred.

In order to prove that our simulation relation is maintained by each step of the algorithm, we must use certain invariants of the DTML model. These invariants are similar to the corresponding invariants used in a proof of the original TML algorithm for the conventional volatile RAM model (see [10] for details). For example, our invariants imply that there is at most one writing transaction, and there is no such transaction when *glb* is even. The main additional invariant that we use for DTML constrains the possible differences between volatile and persistent memory: volatile and persistent memory are identical except for any location that has been written by a writer or by the recovery procedure but not yet flushed. This simple invariant combined with the global relation is enough to prove that the memory state after each crash is correct. Our DTML invariants have been verified in Isabelle, and can be found in the Isabelle files.

6 Related Work

Although there is existing research on extending the definition of linearizability to durable systems, there is comparatively less work on extending other notions of transactional memory correctness such as, but not limited to, opacity to durable systems. Various systems attempt to achieve atomic durability, transform general objects to persistent objects and provide a secure interface of persistent memory. The above goals usually require the use of logging which can be software or hardware based. Raad et al have proposed a notion of durable serialisability under relaxed memory [34], but this model does not handle aborted transactions.

ATLAS [6] is a software system that provides durability semantics for NVRAM with lock-base multithreaded code. The system ensures that the outermost critical sections, which are protected by one or more mutexes, are failure-atomic by identifying

Failure Atomic Sections (FASEs). These sections ensure that, if at least one update that occurs to a persistent location inside a FASE is durable, then all the updates inside the session are durable. Furthermore, like DTML, ATLAS keeps an persistent undo log, that tracks the synchronisation operations and persistent stores, and allows the recovery of rollback FASEs that were interrupted by crashes.

Koburn et al. [7] integrate persistent objects into conventional programs, and furthermore seek to prevent safety bugs that occur in predominantly persistent memory models, such as multiple frees, pointer errors, and locking errors. This is done by implementing NV-heaps, an interface to the NVRAM based on ACID transactions that guarantees safety and provides reasoning about the order in which changes to the data structures should become permanent. NV-heaps only handle updates to persistent memory inside transactions and critical sections. Other systems based on persistent ACID transactions include Mnemosyne [37], Stasis [36] and BerkeleyDB [33].

Ben-David et al. [4] developed a system that can transform programs that consist of read, write and CAS operations in shared memory, to persistent memory. The system aims to create concurrent algorithms that guarantee consistency after a fault. This is done by introducing checkpoints, which record the current state of the execution and from which the execution can be continued after a crash. Two consecutive checkpoints form a *capsule*, and if a crash occurs inside a capsule, program execution is continued from the previous capsule boundary. We have not applied this technique to develop DTML, but it would be interesting to develop and optimise capsules in an STM context.

Mnemosyne [37] provides a low-level interface to persistent memory with high-level transactions based on TinySTM [18] and a redo log that is purposely chosen to reduce ordering constraints. The log is flushed at the commit of each transaction. As a result, the memory locations that are written to by the transaction remain unmodified until commit. Each read operation checks whether data has been modified and if so, returns the buffered value instead of the value from the memory. The size of the log increases proportionally to the size of the transaction, potentially making the checking time consuming.

Hardware based durable transactional memory has also been proposed [24] with hardware support for redo logging [25]. Other indicative hardware systems help implement atomic durability either by performing accelerated ordering or by performing the logging operation are [27, 31].

7 Conclusions

In this paper we have defined durable opacity, a new correctness condition for STMs, inspired by durable linearizability [22] for concurrent objects. The condition assumes a history with crashes such that in-flight transactions are aborted (i.e., do not continue) after a crash takes place, and simply requires that the history satisfies opacity [19,20] after the crashes are removed. This is a strong notion of correctness but ensures safety for STMs in the same way that durable linearizability [22] ensures safety for concurrent objects. It is already known that TMS1 [15], which is a weaker condition than opacity [26] is sufficient for contextual refinement [3]; therefore we conjecture that durable opacity can provide similar guarantees in a non-volatile context. For concurrent objects, more

relaxed notions such as buffered durable linearizability [22] have also been proposed, which requires causally related operations to be committed in order, but real-time order need not be maintained. Such notions could also be considered in a transactional memory setting [16], but the precise specification of such a condition lies outside the scope of this paper.

To verify durable opacity, we have developed DTMS2, an operational characterisation that extends the TMS2 specification with a crash operation. We establish that all traces of DTMS2 are durably opaque, which makes it possible to prove durable opacity by showing refinement between an implementation and DTMS2. We develop a durably opaque example implementation, DTML, which extends TML [8] with a persistent undo log, and associated modifications such as the introduction of a recovery operation. Finally, we prove durable opacity of DTML by establishing a refinement between it and DTMS2. This proof has been fully mechanised in Isabelle.

Our focus has been on the formalisation of durable opacity and the development of an example algorithm and verification technique. Future work will consider alternative implementations of the algorithm, e.g., using a persistent set [38], or thread-local undo logs [23]. develop and implement a logging mechanism based on undo and redo log properties named JUSTDO logging. This mechanism aims to reduce the memory size of log entries while preserving data integrity after crash occurrences. Unlike optimistic transactions [6], JUSTDO logging resumes the execution of interrupted FASEs to their last store instruction, and then executes them until completion. A small log is maintained for each thread, that records its most recent store within a FASE, simplifying the log management and reduce the memory requirements. Future work will also consider weakly consistent memory models building on existing works integrating persistency semantics with hardware memory models [34, 35].

References

1. Armstrong, A., Dongol, B.: Modularising opacity verification for hybrid transactional memory. In: Bouajjani, A., Silva, A. (eds.) FORTE. LNCS, vol. 10321, pp. 33–49. Springer (2017)
2. Armstrong, A., Dongol, B., Doherty, S.: Proving opacity via linearizability: A sound and complete method. In: Bouajjani, A., Silva, A. (eds.) FORTE. LNCS, vol. 10321, pp. 50–66. Springer (2017)
3. Attiya, H., Gotsman, A., Hans, S., Rinetzky, N.: Safety of live transactions in transactional memory: TMS is necessary and sufficient. In: Kuhn, F. (ed.) DISC. LNCS, vol. 8784, pp. 376–390. Springer (2014)
4. Ben-David, N., Blelloch, G.E., Friedman, M., Wei, Y.: Delay-free concurrency on faulty persistent memory. In: The 31st ACM Symposium on Parallelism in Algorithms and Architectures. pp. 253–264 (2019)
5. Bila, E., Doherty, S., Dongol, B., Derrick, J., Schellhorn, G., Wehrheim, H.: Defining and verifying durable opacity: Correctness for persistent software transactional memory (2020), `https://arxiv.org/abs/2004.08200`
6. Chakrabarti, D.R., Boehm, H.J., Bhandari, K.: Atlas: Leveraging locks for non-volatile memory consistency. ACM SIGPLAN Notices **49**(10), 433–452 (2014)
7. Coburn, J., Caulfield, A.M., Akel, A., Grupp, L.M., Gupta, R.K., Jhala, R., Swanson, S.: Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. ACM SIGARCH Computer Architecture News **39**(1), 105–118 (2011)

8. Dalessandro, L., Dice, D., Scott, M.L., Shavit, N., Spear, M.F.: Transactional mutex locks. In: D'Ambra, P., Guarracino, M.R., Talia, D. (eds.) Euro-Par (2). LNCS, vol. 6272, pp. 2–13. Springer (2010)
9. Dalessandro, L., Spear, M.F., Scott, M.L.: Norec: streamlining STM by abolishing ownership records. In: Govindarajan, R., Padua, D.A., Hall, M.W. (eds.) PPoPP. pp. 67–78. ACM (2010)
10. Derrick, J., Doherty, S., Dongol, B., Schellhorn, G., Travkin, O., Wehrheim, H.: Mechanized proofs of opacity: a comparison of two techniques. Formal Asp. Comput. **30**(5), 597–625 (2018)
11. Derrick, J., Doherty, S., Dongol, B., Schellhorn, G., Wehrheim, H.: Verifying correctness of persistent concurrent data structures. In: FM. Lecture Notes in Computer Science, vol. 11800, pp. 179–195. Springer (2019)
12. Derrick, J., Dongol, B., Schellhorn, G., Travkin, O., Wehrheim, H.: Verifying opacity of a transactional mutex lock. In: FM. LNCS, vol. 9109, pp. 161–177. Springer (2015)
13. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Dolev, S. (ed.) DISC. LNCS, vol. 4167, pp. 194–208. Springer (2006)
14. Doherty, S., Dongol, B., Derrick, J., Schellhorn, G., Wehrheim, H.: Proving opacity of a pessimistic STM. In: Fatourou, P., Jiménez, E., Pedone, F. (eds.) OPODIS. LIPIcs, vol. 70, pp. 35:1–35:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016)
15. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Towards formally specifying and verifying transactional memory. Formal Asp. Comput. **25**(5), 769–799 (2013)
16. Dongol, B., Jagadeesan, R., Riely, J.: Transactions in relaxed memory architectures. PACMPL **2**(POPL), 18:1–18:29 (2018)
17. Dongol, B., Derrick, J.: Verifying linearisability: A comparative survey. ACM Comput. Surv. **48**(2), 19:1–19:43 (2015)
18. Felber, P., Fetzer, C., Riegel, T.: Dynamic performance tuning of word-based software transactional memory. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. pp. 237–246 (2008)
19. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: Chatterjee, S., Scott, M.L. (eds.) PPOPP. pp. 175–184. ACM (2008)
20. Guerraoui, R., Kapalka, M.: Principles of Transactional Memory. Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers (2010)
21. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM TOPLAS **12**(3), 463–492 (1990)
22. Izraelevitz, J., Mendes, H., Scott, M.L.: Linearizability of persistent memory objects under a full-system-crash failure model. In: Gavoille, C., Ilcinkas, D. (eds.) DISC. LNCS, vol. 9888, pp. 313–327. Springer (2016)
23. Izraelevitz, J., Kelly, T., Kolli, A.: Failure-atomic persistent memory updates via justdo logging. ACM SIGARCH Computer Architecture News **44**(2), 427–442 (2016)
24. Joshi, A., Nagarajan, V., Cintra, M., Viglas, S.: Dhtm: Durable hardware transactional memory. In: 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA). pp. 452–465. IEEE (2018)
25. Joshi, A., Nagarajan, V., Viglas, S., Cintra, M.: Atom: Atomic durability in non-volatile memory through hardware logging. In: 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA). pp. 361–372. IEEE (2017)
26. Lesani, M., Luchangco, V., Moir, M.: Putting opacity in its place. In: Workshop on the Theory of Transactional Memory (2012)
27. Lu, Y., Shu, J., Sun, L., Mutlu, O.: Loose-ordering consistency for persistent memory. In: 2014 IEEE 32nd International Conference on Computer Design (ICCD). pp. 216–223. IEEE (2014)
28. Lynch, N., Vaandrager, F.: Forward and backward simulations. Information and Computation **121**(2), 214 – 233 (1995)

29. Lynch, N.A., Tuttle, M.R.: Hierarchical correctness proofs for distributed algorithms. In: PODC. pp. 137–151. ACM, New York, NY, USA (1987)
30. Müller, O.: I/O Automata and beyond: Temporal logic and abstraction in Isabelle. In: Grundy, J., Newey, M. (eds.) TPHOLs. pp. 331–348. Springer Berlin Heidelberg, Berlin, Heidelberg (1998)
31. Nalli, S., Haria, S., Hill, M.D., Swift, M.M., Volos, H., Keeton, K.: An analysis of persistent memory use with whisper. ACM SIGPLAN Notices **52**(4), 135–148 (2017)
32. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
33. Olson, M.A., Bostic, K., Seltzer, M.I.: Berkeley db. In: USENIX Annual Technical Conference, FREENIX Track. pp. 183–191 (1999)
34. Raad, A., Wickerson, J., Vafeiadis, V.: Weak persistency semantics from the ground up: formalising the persistency semantics of armv8 and transactional models. PACMPL **3**(OOPSLA), 135:1–135:27 (2019)
35. Raad, A., Vafeiadis, V.: Persistence semantics for weak memory: integrating epoch persistency with the TSO memory model. PACMPL **2**(OOPSLA), 137:1–137:27 (2018)
36. Sears, R., Brewer, E., Brewer, E., Brewer, E.: Stasis: flexible transactional storage. In: Proceedings of the 7th symposium on Operating systems design and implementation. pp. 29–44. USENIX Association (2006)
37. Volos, H., Tack, A.J., Swift, M.M.: Mnemosyne: Lightweight persistent memory. ACM SIGARCH Computer Architecture News **39**(1), 91–104 (2011)
38. Zuriel, Y., Friedman, M., Sheffi, G., Cohen, N., Petrank, E.: Efficient lock-free durable sets. PACMPL **3**(OOPSLA), 128:1–128:26 (2019)